Un Lenguaje para la Especificación y Validación de Arquitecturas de Software

Tesis Doctoral

Presentada por
D. Carlos Canal Velasco
para optar al grado de
Doctor Ingeniero en Informática

Dirigida por el Dr.
D. José María Troya Linero,
Catedrático de Universidad del
Área de Lenguajes y Sistemas Informáticos
y por el Dr.
D. Ernesto Pimentel Sánchez,
Titular de Universidad del
Área de Lenguajes y Sistemas Informáticos

Málaga, Diciembre de 2000

D. José María Troya Linero, Catedrático de Universidad del Área de Lenguajes y Sistemas Informáticos de la E.T.S. de Ingeniería Informática de la Universidad de Málaga, y

D. Ernesto Pimentel Sánchez, Titular de Universidad del Área de Lenguajes y Sistemas Informáticos de la E.T.S. de Ingeniería Informática de la Universidad de Málaga,

Certifican

Que D. José Carlos Canal Velasco, Licenciado en Informática, ha realizado en el Departamento de Lenguajes y Ciencias de la Computación de la Universidad de Málaga, bajo nuestra dirección, el trabajo de investigación correspondiente a su Tesis Doctoral titulada

Un Lenguaje para la Especificación y Validación de Arquitecturas de Software

Revisado el presente trabajo, estimamos que puede ser presentado al tribunal que ha de juzgarlo, y autorizamos la presentación de esta Tesis Doctoral en la Universidad de Málaga.

Málaga, Diciembre de 2000

Fdo.: José María Troya Linero
Catedrático de Universidad del Área
de Lenguajes y Sistemas Informáticos.

Fdo.: Ernesto Pimentel Sánchez
Titular de Universidad del Área
de Lenguajes y Sistemas Informáticos.

La naturaleza esconde, no revela. Todo ese universo luminoso y oscuro a cuya suma de energías llamamos naturaleza es una máscara, un vestido y un sueño. La naturaleza no es Isis, sino el velo de Isis.

Fernando Pessoa

Agradecimientos

La realización de una tesis doctoral es una aventura vital de carácter personal, pero que al igual que la propia vida, no puede llegar a buen puerto sin la ayuda, el aliento y el afecto de quienes nos rodean.

Sirva esta página para dar las gracias a todos los que, de un modo u otro, han contribuido a la culminación de esta aventura: familiares, compañeros, amigos... Resulta imposible hacer aquí una enumeración exhaustiva. No obstante, quiero dedicar especialmente este trabajo a las siguientes personas, como muestra de mi agradecimiento hacia ellas:

En primer lugar, debo mencionar a mis directores —José María y Ernesto—, quienes depositaron en mí su confianza para la realización de este trabajo. Quiero daros las gracias por vuestros sabios consejos y por la dedicación que me habéis prestado a lo largo de todo este tiempo.

También a Antonio, compañero de fatigas, de estudio y discusiones sobre éste y otros muchos temas. Gracias por tu paciencia y tus valiosos comentarios a la hora de revisar esta memoria.

Una mención muy especial la merece Lidia, compañera y amiga en todo y a pesar de todo. ¿Qué puedo decir respecto a ti? Cualquier cosa sería poco. Gracias por ser como eres.

A Antonio, Carmen y Carmen por los buenos y malos momentos que hemos pasado juntos. Gracias por estar siempre ahí.

Por último, a Dani y a Alfonso, por vuestra amistad y por todo el apoyo que me habéis ofrecido en los momentos difíciles.

A todos vosotros, gracias.

Portada: Iglesia de San Miguel de Lillo.
Monte Naranco (Asturias).
Estilo prerrománico asturiano. Siglo IX.

Índice General

Prólogo

A medida que aumenta la complejidad de los sistemas de software surgen nuevos aspectos del desarrollo de aplicaciones que hasta ese momento no se habían tenido en cuenta, al menos de una forma explícita. De este modo, el proceso de desarrollo se ha ido convirtiendo gradualmente en una labor de ingeniería, en la que nuevas tareas, como la elaboración de especificaciones, el diseño del sistema, la construcción de prototipos, la integración y pruebas, la gestión de la configuración y otras muchas han ido cobrando importancia frente a las labores de programación que eran las que primaban en un inicio. La Ingeniería del Software ha ido respondiendo a esta situación con el desarrollo de nuevos modelos, notaciones, técnicas y métodos.

Dentro de esta tendencia se encuadra el creciente interés por los aspectos arquitectónicos del software del que estamos siendo testigos en los últimos tiempos. Estos aspectos se refieren a todo lo relativo a la estructura de alto nivel de los sistemas: su organización en subsistemas y la relación entre éstos; la construcción de aplicaciones vista como una actividad fundamentalmente *composicional* en la que se reutilizan elementos creados posiblemente por terceros; el desarrollo de familias de productos caracterizadas por presentar una arquitectura común; el mantenimiento y la evolución entendidos como sustitución de unos componentes por otros dentro de un marco arquitectónico, etc. En efecto, un aspecto crítico a la hora de desarrollar sistemas de software complejos es el diseño de su arquitectura, representada como un conjunto de elementos computacionales y de datos interrelacionados de un modo determinado.

Este interés no debe tomarse como algo aislado, sino que está integrado en una tendencia más general que ha llevado al establecimiento de lo que ya se denomina como Ingeniería del Software basada en Componentes. Fenómenos como el uso generalizado de la Web como medio para la comunicación y el intercambio global demandan el desarrollo de nuevas tecnologías para la construcción de aplicaciones abiertas y distribuidas. Estas aplicaciones tienen una estructura compleja y dinámica, formada por la interconexión de una gran cantidad de componentes. Las características específicas de estos sistemas están teniendo un enorme impacto en la forma en que el software es producido y comercializado. Desde este punto de vista, el objetivo de la Ingeniería del Software basada en Componentes es la creación de una colección de componentes de software reutilizables, de forma que el desarrollo de nuevas aplicaciones consista básicamente en la selección, adaptación y composición de componentes, en lugar de implementar la aplicación desde el principio.

Si bien es importante la reutilización de componentes, no lo es menos el poder reutilizar la propia estructura o arquitectura de la aplicación. Desde este punto de vista, se ha caracterizado recientemente la Arquitectura del Software como un campo específico de estudio para los investigadores e ingenieros del software. Su ámbito se centra en el nivel del proceso de diseño en el que se deciden las propiedades estructurales del sistema, es decir, aquéllas derivadas de la forma en la que se combinan sus diferentes partes y que no pueden ser tratadas de forma adecuada dentro de los módulos o componentes que forman el sistema.

No obstante, el campo de estudio de la Arquitectura del Software no es algo nuevo. Los sistemas de software han tenido arquitectura desde que el primer programa fue dividido en módulos, y los programadores han sido responsables de establecer las interacciones entre dichos módulos y lograr así determinadas propiedades globales para sus sistemas. Históricamente, la arquitectura de los sistemas ha sido desarrollada y utilizada de forma implícita, en muchos casos como mero accidente durante el proceso de implementación, o como herencia de sistemas anteriores. Los profesionales del desarrollo de software han adoptado a menudo uno o varios patrones arquitectónicos como estrategia para la organización de sus aplicaciones, pero han utilizado estos patrones de manera informal, sin reflejarlos de manera explícita en el sistema resultante.

De este modo, la descripción de los aspectos arquitectónicos del software ha estado tradicionalmente limitada al uso de ciertas expresiones, tales como arquitectura cliente/servidor, arquitectura en capas, etc., por lo general acompañadas con diagramas informales. Estas descripciones carecen de un significado preciso, lo que limita de manera drástica su utilidad, impidiendo, por ejemplo, cualquier análisis de las propiedades de los sistemas así descritos. Lógicamente, este tipo de representaciones no es el adecuado para dar a la Arquitectura del Software el rango que se merece. Existe una clara necesidad de notaciones de alto nivel, específicamente orientadas al problema de describir la arquitectura de los sistemas de software.

La importancia de representar de forma explícita la arquitectura de los sistemas de software es evidente. En primer lugar, estas representaciones elevan el nivel de abstracción, facilitando la comprensión de los sistemas de software complejos. En segundo lugar, hacen que aumenten las posibilidades de reutilizar tanto la arquitectura como los componentes que aparecen en ella. Por último, si la notación utilizada tiene una base formal adecuada, es posible analizar la arquitectura del sistema, determinando cuáles son sus propiedades aún antes de construirlo.

En los últimos años se han ido proponiendo toda una serie de lenguajes para la descripción de la arquitectura del software. Aunque la mayoría de estos lenguajes de primera generación están aún en desarrollo o tienen carácter experimental, son buenos ejemplos de la búsqueda de las estructuras lingüísticas necesarias para describir las propiedades arquitectónicas de los sistemas de software.

Objetivos

El presente proyecto de tesis se enmarca dentro de este contexto. Su principal aportación es la definición de LEDA, un lenguaje de especificación para la descripción y validación de la arquitectura de los sistemas de software. Entre los objetivos planteados a la hora de realizar este trabajo debemos destacar los siguientes:

- El lenguaje debe abordar la descripción de la arquitectura del software siguiendo para ello un enfoque composicional, de forma que los sistemas se construyan mediante la interconexión de componentes más sencillos.

- La especificación de los componentes debe limitarse a su interfaz, en la que se describirá toda la información de interés desde un punto de vista arquitectónico, pero ocultando al mismo tiempo detalles de implementación que sean irrelevantes a este nivel.

- La propuesta ha de tener un fundamento formal, adecuado a las necesidades de la Arquitectura del Software, y que permita el análisis de los sistemas descritos, con objeto de validar sus propiedades.

- Ha de ser posible la descripción de sistemas dinámicos, es decir, que presenten una configuración o una topología de comunicación cambiantes.

- El lenguaje dispondrá de mecanismos que faciliten la evolución del sistema, tanto durante la especificación del mismo —que será considerada como un proceso de refinamiento—, como una vez implementado, con vistas a adaptarlo a cambios en los requisitos.

- Del mismo modo, han de incorporarse mecanismos que faciliten la reutilización, tanto de los componentes como de la arquitectura.

- La propuesta no debe limitarse únicamente a la especificación y análisis de la arquitectura, sino que tiene que facilitar la implementación del producto final, ofreciendo una guía para el proceso de desarrollo.

- Las especificaciones deben poder utilizarse tanto para la simulación de los sistemas descritos como para la generación de prototipos a partir de las mismas, con objeto de automatizar en la medida de lo posible el desarrollo del sistema.

Estructura de la memoria

Esta memoria está organizada en cinco capítulos. En el primero de ellos se ofrece una visión general de la Arquitectura del Software y del estado del arte en este campo, poniendo especial énfasis, como es lógico, en los lenguajes de descripción de arquitectura, para lo que se comentan las propuestas más significativas realizadas hasta la fecha. Además, se aborda el estudio de las relaciones entre la Arquitectura del Software y otros campos y propuestas relacionados, como son la Ingeniería del Software basada en Componentes, los patrones de diseño o el Lenguaje Unificado de Modelado. Así mismo, este capítulo contiene una revisión de los formalismos utilizados en las diferentes propuestas sobre Arquitectura del Software, incluyendo entre ellos al cálculo π, que es la base formal utilizada en el presente trabajo.

El Capítulo 2 se dedica a los fundamentos formales de este trabajo. En él se desarrolla un estudio más a fondo del cálculo π y de sus cualidades para la descripción de la arquitectura del software. Se definen además los conceptos de rol y de arquitectura en el contexto del cálculo π y se presentan relaciones de compatibilidad, herencia y extensión entre roles, demostrándose diversos resultados a partir de las mismas, como son la composición y la sustitución con éxito de componentes en el contexto de una arquitectura.

A continuación, en el Capítulo 3 se presenta LEDA, el lenguaje de descripción de arquitectura objeto de este trabajo. Las características más significativas del lenguaje se describen con ayuda de numerosos ejemplos. En este capítulo se esboza también la semántica del lenguaje en términos del cálculo π. Por último, un estudio de caso muestra cómo utilizar el lenguaje para la especificación de un sistema distribuido de subastas, ejemplo típico de aplicación desarrollada sobre Internet.

El Capítulo 4 trata de los aspectos metodológicos del desarrollo de sistemas a partir de su especificación en LEDA. En él se describe cómo obtener un prototipo en un lenguaje orientado a objetos a partir de la especificación, y cómo este prototipo sirve de base para el desarrollo iterativo e incremental del sistema. En particular, el generador produce código Java, aunque nuestra propuesta puede aplicarse a cualquier otro lenguaje orientado a objetos. El núcleo del generador de código está formado por un conjunto de clases que modelan procesos, enlaces y acciones, por lo que puede considerarse como un intérprete del cálculo π en Java.

Por último, el Capítulo 5 contiene las conclusiones de este trabajo, así como una serie de líneas futuras de continuación del mismo. Esta memoria se completa con tres apéndices, que contienen la sintaxis BNF y la notación gráfica del lenguaje, así como el código completo generado para el estudio de caso del Capítulo 3.

Capítulo 1

Introducción

En este capítulo pretendemos ofrecer una visión general de lo que es la Arquitectura del Software, su objeto de estudio, su ámbito de actuación y las líneas de investigación actualmente existentes en este campo. También abordamos brevemente, en función de su relación con la Arquitectura del Software, otros campos de estudio, como son la Ingeniería del Software basada en Componentes o los Métodos Formales para el Desarrollo de Software.

A pesar de tener un origen distinto —más *académico* en el caso de la Arquitectura del Software y más *industrial* en el de la Ingeniería del Software basada en Componentes— la opinión generalizada es que componentes y arquitectura están íntimamente ligados [Kruchten, 1995, Brown y Wallnau, 1998, Szyperski, 2000]. Los integrantes de una y otra corriente comparten una visión *composicional* del software, centrada en la noción de componente como unidad elemental, y no sólo para las etapas iniciales del desarrollo, es decir, durante la especificación y el diseño, sino también durante la implementación (que se convierte en ensamblado de componentes reutilizables dentro de algún marco arquitectónico), y evolución del sistema (que consiste entonces en el reemplazamiento de componentes y la reconfiguración de la arquitectura del sistema). Por otro lado, ambas aproximaciones se basan en el concepto de *interfaz* como medio para describir los componentes, haciendo explícita la funcionalidad que ofrecen y sus dependencias de contexto, y ocultando los detalles internos de su implementación. La composición de componentes para formar sistemas mayores se realiza por medio de la conexión de estas interfaces. Como veremos más adelante, los llamados *marcos de trabajo* son el exponente más claro de esta estrecha relación entre componentes y arquitectura.

Por estos motivos, comenzaremos este Capítulo hablando de Ingeniería del Software basada en Componentes, como disciplina más amplia en la que se encuadraría la Arquitectura del Software.

1.1 Ingeniería del Software basada en Componentes

La Ingeniería del Software basada en Componentes (*Component-Based Software Engineering*, CBSE) es una disciplina reciente, que se encuadra dentro de la Ingeniería del Software, y a la se está prestando una atención cada vez mayor. Este interés se debe a la concurrencia de diversos factores, entre los que debemos citar la aparición y progresiva utilización de las plataformas de componentes, que facilitan el uso de la llamada tecnología de componentes y hacen posible el desarrollo de verdaderos sistemas abiertos y de aplicaciones *plug-and-play* [Krieger y Adler, 1998]. Por otro lado, no es tampoco ajena a este fenómeno la continua expansión de la WWW (*World-*

Wide Web) como marco global para el intercambio de información y servicios, incluidos los comerciales, con la consiguiente necesidad de aplicaciones capaces de realizar este intercambio.

Los antecedentes más significativos de las plataformas de componentes los podemos establecer en DCE y CORBA, desarrolladas a iniciativa de los consorcios OSF (*Open Software Foundation*) y OMG (*Object Management Group*), respectivamente. Aunque DCE (*Distributed Computing Environment*) [OSF, 1994] tuvo en un principio una buena acogida por parte de la industria, algunas de las limitaciones que presenta (no es orientada a objetos, no define servicios de transacciones y sólo permite invocaciones estáticas, no dinámicas) le han hecho perder gran parte de su cuota de mercado en el desarrollo de aplicaciones distribuidas. Sin embargo, la especificación de la arquitectura CORBA (*Common Object Request Broker Architecture*) [OMG, 1999, Vinoski, 1998], a pesar de definirse como una plataforma de *objetos* distribuidos, asienta los principios básicos de la tecnología de componentes, habiendo consolidado en gran medida su posición. Posteriormente han aparecido diversas plataformas, entre las que debemos citar, por su importancia, COM (*Component Object Model*), desarrollada por Microsoft [Rogerson, 1997, Box, 1998] y JavaBeans, desarrollada por Sun [Sun Microsystems, 1997]. Estas tres plataformas constituyen los estándares *de facto* en este terreno, aunque su continua evolución ha dado lugar a numerosos modelos y productos derivados, entre los que se encuentran CCM (*CORBA Component Model*), EJB (*Enterprise Java Beans*), DCOM (*Distributed COM*) [Microsoft, 1996], y la recientemente anunciada .NET, también de Microsoft.

1.1.1 Sistemas de componentes

La CBSE surge, por tanto, de la necesidad de dotar de un soporte metodológico al desarrollo de aplicaciones utilizando la tecnología de componentes y las plataformas citadas. Su objeto es la construcción de aplicaciones mediante el ensamblado de componentes ya existentes, en lugar de partir de cero. En este contexto, el desarrollo de sistemas de software implica la reutilización de componentes prefabricados —los llamados COTS (*Commercial Off-The-Shelf*)—, posiblemente desarrollados por terceras partes y no de forma específica para una aplicación concreta, sino pensando en su integración en diversos sistemas, de manera que se cree un verdadero *mercado de componentes* [Brown y Wallnau, 1998]. Uno de los sueños que siempre ha tenido la Ingeniería del Software es el de contar con un mercado global de componentes, al igual que ocurre con otras ingenierías, o incluso con el hardware.

Entre las características más significativas de este tipo de sistemas podemos indicar que son:

- **Distribuidos.** Formados por componentes que residen en diversas máquinas integrantes de una red.
- **Heterogéneos.** No sólo en cuanto a la plataforma hardware y sistema operativo de las máquinas donde se ejecutan los componentes, sino también en cuanto a los lenguajes de implementación utilizados para desarrollar dichos componentes, y en cuanto a quiénes han implementado cada uno de ellos.
- **Independientemente extensibles.** De forma que puedan ser modificados y ampliados mediante la integración en el sistema de nuevos componentes y su conexión con los ya existentes. Los nuevos componentes pueden haber sido desarrollados por terceras partes, sin conocimiento entre ellas, ni del sistema actualmente existente.
- **Dinámicos.** Sujetos a continua evolución, no sólo por la integración de nuevos componentes sino también por la desaparición o sustitución de alguno de los ya existentes y por la reconfiguración de las relaciones entre ellos.

Como consecuencia de las características citadas, el desarrollo de aplicaciones para este tipo de sistemas se ve afectado por una serie de problemas específicos, como son la gestión de la evolución del propio sistema y de sus componentes, la falta de una visión global del sistema, la dificultad para garantizar la seguridad y confidencialidad de los mensajes, etc.

Estas y otras cuestiones son las que han puesto de manifiesto la necesidad de nuevos modelos y métodos de desarrollo de software, pues la programación tradicional, y en particular la programación orientada a objetos, se ha visto incapaz de tratarlos de forma natural. A continuación trataremos de aclarar la noción de componente, para seguidamente establecer la diferencia entre componentes y objetos.

1.1.2 Noción de componente

Como es de esperar, debido a lo reciente de este campo, no existe aún un consenso en la comunidad del software acerca de lo que es un componente ni en cuáles son las características significativas que diferencian a los componentes de otras unidades de software.

Así, para Wallnau, un componente es una unidad de software que presenta una funcionalidad y complejidad significativas y que puede ser considerado como una caja negra con vistas a su integración con otros componentes [Wallnau et al., 1997], haciendo hincapié en su definición en que los componentes son entidades de grano grueso, y en que, lógicamente, están orientados a la composición.

Kruchten coincide en afirmar que un componente es una parte no trivial, casi independiente y reemplazable de un sistema [Kruchten, 1995]. Añade además que un componente lleva a cabo una función claramente establecida y en el contexto de una arquitectura bien definida, y que proporciona la realización física de un conjunto de interfaces, a las cuales se ajusta.

Sin embargo, Nierstrasz y Meijler indican que un componente puede ser tanto de grano grueso como de grano fino [Nierstrasz y Meijler, 1994]. Ejemplos de componentes de grano fino pueden ser los *mixins*, cuya composición permite definir, extender y especializar clases de objetos, o también los botones y ventanas de diálogo que se usan habitualmente en las plataformas COM o JavaBeans. Por el contrario, ejemplos de componentes de grano grueso serían una hoja de cálculo, que puede ser integrada en un documento de texto, o algunos programas de Unix, como *grep* que pueden ser combinados unos con otros mediante tuberías para realizar complejos procesamientos de texto.

Por su parte, Szyperski afirma que un componente es una unidad binaria de composición, carente de estado interno, y que está caracterizado por una serie de interfaces que establecen un contrato entre el componente y su entorno. En estas interfaces se debe declarar tanto la funcionalidad que el componente ofrece al exterior, como también todas las dependencias de contexto del componente, es decir, qué funcionalidad requiere para su funcionamiento, o qué requisitos determinan las necesidades del componente en cuanto a recursos, como por ejemplo la plataforma de ejecución que precisa para funcionar. Descrito de esta forma un componente de software puede ser instalado de forma independiente y está sujeto a composición por terceras partes [Szyperski, 1998].

En cambio, Meyer afirma que el carácter fundamental que define a un componente no es su tamaño o su carácter binario o fuente, sino el hecho de que se trata de *software orientado al cliente*, es decir, que permita ser utilizado por otros elementos de software (sus clientes) sin que sea necesaria la intervención de los autores del componente, y sin que estos autores necesiten saber nada acerca de quiénes serán sus clientes ni los autores de éstos. Para ello, el componente debe incluir una especificación completa de su funcionalidad y de todas las

dependencias que presente (plataforma hardware y software, otros componentes, etc.), y debe poder ser integrado de forma rápida y sencilla en un sistema, partiendo únicamente de la base de dicha especificación [Meyer, 1999].

A pesar de estas discrepancias en cuanto al tamaño, complejidad, e incluso carácter de los componentes, una cosa parece estar clara: *los componentes son unidades de composición*, y su existencia sólo tiene sentido como parte de un entorno o plataforma de componentes. En efecto, si tomamos como ejemplo un equipo musical o un mueble modular, los elementos que forman estos sistemas son componentes debido exclusivamente a que han sido diseñados para ser compuestos unos con otros, permitiendo múltiples combinaciones y resultados. Lo mismo ocurre con el software. Considérense, por ejemplo, un componente *botón* de una interfaz gráfica de usuario. Su valor reside no tanto en la funcionalidad que proporciona como en el hecho de que ha sido diseñado para colaborar con otros componentes de forma consistente para construir interfaces de usuario complejas.

Por tanto, un componente individual que no pertenezca a un sistema de componentes de este tipo o que no pueda ser compuesto con otros es una contradicción en sí mismo. Los componentes no pueden existir fuera de un entorno de composición definido. Nierstrasz y Dami hacen hincapié en estos aspectos al afirmar que un componente no se desarrolla de manera aislada, sino como elemento de un marco de trabajo composicional que proporciona *(i)* una biblioteca de componentes, *(ii)* una arquitectura de software reutilizable en la que encajan dichos componentes, y *(iii)* un mecanismo de conexión que permita su composición [Nierstrasz y Dami, 1995].

Como vemos, disponer de componentes no es suficiente, a menos que seamos capaces de ensamblarlos y reutilizarlos. Y reutilizar un componente no significa usarlo más de una vez, sino que implica la capacidad del componente para ser utilizado en contextos distintos a aquéllos para los que fue diseñado. El uso de componentes de software y de los mecanismos de composición adecuados proporcionan los medios para la reutilización sistemática del software [Shaw, 1995]. Además, la composición de componentes de software reutilizables y configurables nos permite abordar el problema de la evolución de los requisitos mediante la sustitución y reconfiguración sólo de las partes del sistema afectadas [Nierstrasz et al., 1992].

1.1.3 Componentes y objetos

La programación orientada a objetos se ha mostrado insuficiente al tratar de aplicar sus técnicas para el desarrollo de aplicaciones basadas en sistemas de componentes. En particular se ha observado que no permite expresar claramente la distinción entre los aspectos composicionales y los meramente computacionales de las aplicaciones, y que hace prevalecer la visión de objeto sobre la de componente, estos últimos como unidades de composición independientemente de las aplicaciones. Por este motivo, surge la llamada *programación orientada a componentes* como una extensión natural de la orientación a objetos [Nierstrasz, 1995b, Szyperski y Pfister, 1996].

A pesar de esta influencia en su origen, la programación orientada de componentes no es una simple evolución de la tecnología de objetos, ni se pueden identificar simplemente los términos 'componente' y 'objeto', sino que existen diferencias sustanciales entre ambas tecnologías [Broy et al., 1998, Henderson-Sellers et al., 1999]. De hecho, la orientación a objetos no utiliza los objetos como unidad elemental del software, al menos durante el desarrollo de las aplicaciones, sino que esta unidad la constituyen las clases, siendo los objetos instancias de una clase y estando su existencia ligada normalmente a un programa en ejecución.

Pero tampoco podemos identificar los conceptos de componente y clase. En primer lugar, aunque un componente pueda estar formado por una única clase, será con más probabilidad

un conjunto de clases, lo que tradicionalmente se ha venido llamando un *módulo*. De hecho, la tecnología de componentes no implica el uso de tecnología de objetos para la implementación de los componentes, pudiendo éstos desarrollarse, por ejemplo, mediante programación estructurada. En segundo lugar, la tecnología de objetos se centra fundamentalmente en los conceptos de ocultación, herencia, polimorfismo y vinculación dinámica, pero deja más de lado otros aspectos como la independencia y la composición tardía. Como consecuencia de ello, la tecnología de objetos ha dado lugar normalmente a la construcción de aplicaciones que no facilitan ni su composición ni la integración en ellas de nuevos componentes. Por último, la tecnología de objetos deja de lado los aspectos económicos y de mercado, por lo que a pesar de las previsiones iniciales, nunca ha desarrollado un mercado de objetos.

Otra diferencia fundamental entre las tecnologías de componentes y de objetos radica en los diferentes mecanismos que ambas proponen para la composición y reutilización, y que están relacionados con la visibilidad de la implementación de los elementos a través de su interfaz.

Según Szyperski, la reutilización en los sistemas de componentes debe basarse exclusivamente en la composición dinámica de interfaces y en el *reenvío* de mensajes entre objetos. Los componentes son cajas negras que ocultan totalmente los detalles de su implementación, siendo su interfaz lo único visible para sus clientes. La reutilización consiste, de esta forma, en la implementación de un componente que proporcione la interfaz requerida por otro o el uso de un componente en función de la interfaz que éste ofrece [Szyperski, 1998].

Por su parte, en la programación orientada a objetos existe también este mecanismo de reutilización por composición; la interfaz sigue encapsulando la implementación, a la vez que limita lo que los clientes de la entidad pueden hacer, pero el mecanismo de herencia permite una notable interferencia. Normalmente, la implementación de una clase está disponible y puede ser estudiada para adquirir un mejor conocimiento de lo que la entidad hace, pero también puede ser manipulada y redefinida por medio de la herencia. El inconveniente es que la reutilización por medio de la herencia provoca un acoplamiento entre la entidad reutilizada y sus clientes conocido como el problema de la *clase base frágil* [Mikhajlov y Sekerinski, 1998], y que dificulta que la entidad reutilizada pueda ser sustituida por una nueva versión de la misma. Probablemente, este reemplazamiento causará problemas en algunos de los clientes/herederos de la clase, debido a que éstos son dependientes de detalles de implementación que han cambiado en la nueva versión.

Las ventajas de la composición frente a la herencia radicarían precisamente en que el reenvío de mensajes no implica un acoplamiento tan fuerte como el que produce el mecanismo de herencia entre clases, por lo que los componentes serán independientes unos de otros. Además, la posibilidad de composición dinámica permite componer y reemplazar los componentes de un sistema incluso durante la ejecución del mismo, lo que no es normalmente posible utilizando herencia, dado el carácter estático de ésta.

1.1.4 Marcos de trabajo composicionales

Uno de los campos en los que la tecnología de componentes se ha mostrado más activa es en el desarrollo de *marcos de trabajo* [Fayad et al., 1999]. Éstos se definen como un diseño reutilizable de todo o parte de un sistema, representado por un conjunto de componentes abstractos, y la forma en la que dichos componentes interactúan. Otra forma de expresar este concepto es que un marco de trabajo es el esqueleto de una aplicación que debe ser adaptado por el programador según sus necesidades concretas. En ambas definiciones podemos ver que un marco de trabajo define el patrón arquitectónico que relaciona los componentes de un sistema [Johnson, 1997].

Podemos establecer dos niveles de representación de un marco de trabajo: por un lado, la especificación de la arquitectura marco, y por el otro, la implementación del marco de trabajo, normalmente utilizando un lenguaje orientado a objetos. Estaremos entonces hablando de marcos de trabajo composicionales y marcos de trabajo de aplicación, respectivamente.

Los marcos de trabajo composicionales representan el grado más alto de reutilización en el desarrollo de software. No sólo los componentes individuales, sino también el diseño arquitectónico es reutilizado en las aplicaciones que se construyen sobre el marco [Pree, 1996]. Las principales ventajas que ofrecen son la reducción del coste en el proceso de desarrollo de aplicaciones para dominios específicos, y la mejora de la calidad del producto final. Son precisamente los marcos de trabajo los que dan sentido a los componentes. El valor de un componente no reside únicamente en la funcionalidad que proporciona, sino en el hecho de que ha sido diseñado para colaborar con otros componentes de forma consistente para construir distintas aplicaciones. Esta composición se realiza por medio de un marco de trabajo [Nierstrasz y Dami, 1995].

Al desarrollo de aplicaciones a partir de un marco de trabajo se le denomina *extensión* del marco. Esta extensión se lleva a cabo en los denominados puntos de entrada (*hot spots*), componentes o procedimientos cuya implementación final depende de la aplicación concreta que queremos desarrollar. Estos puntos de entrada son definidos por el diseñador del marco de trabajo, de manera que sean la forma natural de extensión del mismo [Schmid, 1997]. La extensión puede realizarse de diversas maneras, dando lugar a una clasificación de los marcos de trabajo. De esta forma, podemos hablar de:

- **Marcos de trabajo de caja blanca**, que se extienden mediante herencia, concretamente mediante la implementación de las clases y métodos abstractos definidos como puntos de entrada del marco. En ellos, se tiene acceso al código completo del marco de trabajo y se permite reutilizar la funcionalidad de sus clases mediante herencia y redefinición de métodos.

- **Marcos de trabajo de caja de cristal**, que admiten la inspección de su estructura e implementación, pero no su modificación ni extensión, excepto en los puntos de entrada.

- **Marcos de trabajo de caja gris**, en los que los puntos de entrada no son simplemente métodos abstractos, de los que se declara únicamente su signatura, sino que definen por medio de un lenguaje de especificación de alto nivel los requisitos que deben cumplirse a la hora de implementarlos.

- **Marcos de trabajo de caja negra**, en los que su instanciación se realiza mediante composición y delegación, en lugar de utilizar herencia. Los puntos de entrada están definidos por medio de interfaces que deben de implementar los componentes que extiendan el marco.

Algunos autores postulan que los marcos de trabajo han de ser estrictamente de caja negra [Szyperski, 1998], para evitar las dependencias ya comentadas entre la implementación del marco y la de sus clientes. Otros trabajos apuntan a la necesidad de permitir una mayor visibilidad [Buechi y Weck, 1999], sugiriendo como más adecuado un modelo de caja gris en el que se puedan imponer restricciones a los componentes que implementen los puntos de entrada. Sin embargo, tanto en un caso como en otro, la extensión del marco basada únicamente en la implementación de las interfaces que éste establezca es un ejemplo del llamado *problema de la clarividencia*, puesto que obliga a que todas las interfaces deban estar definidas de antemano, contemplando todos los usos futuros del marco. Este problema se resuelve en los marcos de

trabajo de caja blanca, en los que es posible redefinir componentes y métodos implementados en el mismo. No obstante, el programador que extienda el marco de esta forma debe estar familiarizado con los detalles de implementación de las clases del marco de trabajo, con objeto de definir subclases con sentido [Fayad y Schmidt, 1997].

En definitiva, el desarrollo de aplicaciones a partir de un marco de trabajo se lleva a cabo mediante la extensión del mismo, para lo cual el usuario debe tener información acerca de cuáles son sus puntos de entrada y cómo adaptarlos [Schmid, 1997]. En la literatura al respecto, esto se conoce como el problema de la *documentación del marco de trabajo*. Existen diversas propuestas sobre cómo realizar esta documentación, que sugieren, entre otros, el uso de patrones de diseño [Odenthal y Quibeldey-Cirkel, 1997], diagramas de secuencia de mensajes [Lange y Nakamura, 1995], o contratos de reutilización [Codenie et al., 1997]. No obstante, la representación gráfica del marco, por medio de un lenguaje específico para la descripción de la arquitectura combinado con el uso de entornos visuales que permitan navegar por la arquitectura del marco, es una de las líneas de documentación que más aceptación están teniendo [Pree, 1996].

1.1.5 Componentes y arquitectura

Como hemos podido observar, los componentes no son separables de la arquitectura. Efectivamente, tal es lo que sugieren varias de las definiciones de componente a las que hemos hecho referencia anteriormente. Así mismo, todos los modelos y plataformas de componentes imponen una serie de compromisos sobre dichos componentes, indicando la forma en que se describe su interfaz o qué mecanismos básicos tienen que proporcionar [Brown y Wallnau, 1998]. La interfaz de un componente no sólo muestra su funcionalidad, de forma abstracta, sino que implica además toda una serie de restricciones arquitectónicas ligadas a la plataforma o modelo del que forma parte. En este sentido, la implementación en la plataforma de los aspectos específicos de control del sistema permite la eliminación de estos aspectos del código de los componentes, lo que facilita notablemente su especificación e implementación (centrada ahora únicamente en los aspectos funcionales), favoreciendo al mismo tiempo la reutilización [Szyperski, 2000].

Por otro lado, también hemos visto cómo la relación entre componentes y arquitectura se pone de manifiesto en los marcos de trabajo composicionales. Éstos definen un patrón arquitectónico que describe la estructura de un sistema de software de forma que pueda ser completada mediante la integración en ella de componentes de software, dando así lugar a distintas aplicaciones del marco. Por este motivo, volveremos a ocuparnos brevemente de los marcos de trabajo en la Sección 1.3.3, al hablar de los distintos niveles de abstracción arquitectónica.

En este punto, dejamos el campo más amplio de la tecnología de componentes para centrarnos en el objeto propio de este trabajo: la Arquitectura del Software.

1.2 Arquitectura del Software

A medida que crece la complejidad de las aplicaciones, y que se extiende el uso de sistemas distribuidos y sistemas basados en componentes, los aspectos arquitectónicos del desarrollo de software están recibiendo un interés cada vez mayor, tanto desde la comunidad científica como desde la propia industria del software. El término 'arquitectura' aparece cada vez con mayor frecuencia en la literatura sobre Ingeniería del Software. Para demostrar esto baste señalar que el Proceso Unificado (*Unified Process*) —recientemente presentado como el método

de desarrollo de software asociado al Lenguaje Unificado de Modelado (UML), y que como éste, pretende convertirse en un estándar *de facto*— se define como *centrado en la arquitectura* [Jacobson et al., 1999]. No obstante este creciente interés, sólo recientemente se han abordado de forma explícita los aspectos arquitectónicos, no existiendo aún un claro consenso respecto a qué se entiende por la arquitectura de un sistema de software [Collins-Cope, 2000].

Sin pretender con ello establecer una definición completa ni definitiva, podemos considerar como *arquitectura* la estructura de alto nivel de un sistema de software, lo que incluye sus componentes, las propiedades observables de dichos componentes y las relaciones que se establecen entre ellos [Bass et al., 1998]. Esta definición se centra en aspectos puramente descriptivos, y determina que cualquier sistema de software, o al menos cualquiera que tenga una cierta complejidad, tiene una arquitectura, independientemente de si esta arquitectura está representada en algún lugar de forma explícita, o incluso de si quienes desarrollaron el sistema eran conscientes de ella. Otras definiciones incluyen también aspectos de proceso y, de este modo, Garlan y Perry añaden a la arquitectura de un sistema de software los principios y reglas que gobiernan su diseño y su evolución en el tiempo [Garlan y Perry, 1995].

Teniendo en cuenta todos estos aspectos, e incorporando aún algunos otros, en la ya citada notación UML se define la arquitectura como el conjunto de decisiones significativas acerca de la organización de un sistema de software; la selección de los elementos estructurales a partir de los cuales se compondrá el sistema y sus interfaces, junto con la descripción del comportamiento de dichas interfaces en las colaboraciones que se producen entre los elementos del sistema; la composición de esos elementos estructurales y de comportamiento para formar subsistemas de tamaño cada vez mayor; y el estilo o patrón arquitectónico que guía esta organización: los elementos y sus interfaces, las colaboraciones y su composición [Rumbaugh et al., 1999].

Aparejado a este concepto de arquitectura, surge la Arquitectura del Software, como la disciplina, inscrita dentro de la Ingeniería del Software, que se ocupa del estudio de la arquitectura de los sistemas de software, y también como una de las tareas del proceso de desarrollo, enmarcada dentro de las actividades propias del diseño. Así, de nuevo Garlan y Perry definen la Arquitectura del Software, entendida como disciplina, como el nivel del diseño de software donde se definen la estructura y propiedades globales del sistema [Garlan y Perry, 1995]. Según esto, la Arquitectura del Software se centra en aquellos aspectos del diseño y desarrollo que no pueden tratarse de forma adecuada dentro de los módulos que forman el sistema [Shaw y Garlan, 1996].

Como ya hemos dicho, los sistemas de software siempre han tenido una arquitectura, incluso antes de que nadie mostrase interés por su estudio o por representarla de forma explícita. A lo largo de estos años, los diseñadores han desarrollado un repertorio de métodos, técnicas, patrones y expresiones para estructurar sistemas de software complejos. La Arquitectura del Software simplemente pretende dar rigor y hacer explícito todo este conocimiento.

Como veremos en las secciones siguientes, para lograr este objetivo es necesario desarrollar modelos y lenguajes de descripción de arquitectura, integrar los aspectos arquitectónicos en los métodos de desarrollo, construir herramientas que faciliten el desarrollo de sistemas utilizando estos modelos y notaciones, etc.

1.2.1 Caracterización

La Arquitectura del Software es una disciplina reciente, que está aún dando sus primeros pasos [Szyperski, 2000]. Por este motivo, es necesario caracterizar su ámbito de estudio. Esta caracterización puede realizarse estableciendo una serie de oposiciones o distinciones, que sirven para delimitar los aspectos arquitectónicos frente a otros que no lo son.

Ya hemos hablado de que la Arquitectura del Software se encuadra dentro de la Ingeniería del Software, y en particular del Diseño de Software, por lo que la primera de estas distinciones enfrenta la arquitectura frente a los algoritmos y las estructuras de datos, y pretende fijar el papel de la Arquitectura del Software dentro del proceso de diseño. De este modo, el objeto de la Arquitectura del Software no es el diseño de algoritmos o estructuras de datos, sino la organización a alto nivel de los sistemas de software, incluyendo aspectos como la descripción y análisis de propiedades relativas a su estructura y control global, los protocolos de comunicación y sincronización utilizados, la distribución física del sistema y sus componentes, etc. Junto a éstos, también se presta especial atención a otros aspectos de carácter más general, relacionados con el desarrollo del sistema y su evolución y adaptación al cambio, como son los aspectos de composición, reconfiguración, reutilización, escalabilidad, mantenibilidad, etc. Por tanto, el diseño arquitectónico se centra en lo que tradicionalmente se ha venido llamando diseño preliminar o diseño de alto nivel, frente al nivel de diseño detallado.

La segunda distinción se refiere a la naturaleza de los elementos objeto de estudio y, fundamentalmente, a las relaciones que se establecen entre ellos, y enfrenta las interacciones entre estos componentes con las relaciones de definición y uso que se utilizan para la definición de los módulos de código fuente del sistema. De esta forma, las relaciones de definición/uso modularizan el sistema en función de su código fuente, haciendo explícitas las relaciones de importación y exportación que indican dónde se define y dónde se usa ese código fuente. Sin embargo, desde el punto de vista de la Arquitectura del Software, el sistema se divide en una serie de *componentes*, que no tienen por qué coincidir con los módulos de compilación. Estos componentes realizan computaciones y almacenamiento de datos, y llevan a cabo diversas interacciones unos con otros durante la ejecución del sistema.

La tercera distinción se establece entre los métodos arquitectónicos y los métodos de desarrollo de software (como los orientados a objetos o los estructurados). Mientras el objetivo de estos últimos es proporcionar un camino entre el espacio del problema y el de la solución, la Arquitectura del Software se centra únicamente en el que podríamos denominar como *espacio de los diseños arquitectónicos*, preocupándose de cómo guiar las decisiones a tomar en este espacio —decisiones que se basan en las propiedades de los diferentes diseños arquitectónicos y su capacidad para resolver determinados problemas. No obstante, ambos métodos están estrechamente relacionados y se complementan: detrás de la mayoría de los métodos de desarrollo hay un estilo arquitectónico preferido, mientras que el desarrollo y uso de nuevos estilos arquitectónicos lleva normalmente a la creación de nuevos métodos de desarrollo que exploten sus características.

Finalmente, la última distinción se establece dentro del propio campo de la arquitectura, y enfrenta a los estilos arquitectónicos con las instancias de los mismos. Como veremos en la Sección 1.3, una instancia arquitectónica se refiere a la arquitectura de un sistema concreto, mientras que un estilo arquitectónico define las reglas generales de organización y las restricciones en la forma y la estructura de un grupo numeroso y variado de sistemas de software.

1.2.2 Patrones arquitectónicos y patrones de diseño

En las secciones anteriores hemos caracterizado la Arquitectura del Software como la disciplina que aborda los aspectos estructurales de las aplicaciones durante la fase de diseño preliminar o diseño de alto nivel, y hemos visto que tiene por objeto el estudio de la arquitectura de las aplicaciones y la identificación, definición y uso de *patrones estructurales* para la resolución de las cuestiones que surgen en esta fase del diseño. En esta sección trataremos de las diferencias entre patrones arquitectónicos y patrones de diseño.

Entendemos por patrón una solución probada que se puede aplicar con éxito a un determinado tipo de problemas que aparecen repetidamente en algún campo. El establecimiento de estos patrones comunes es lo que posibilita el aprovechamiento de la experiencia acumulada en el diseño de aplicaciones. Un diseñador experimentado producirá diseños más simples, robustos y generales, y más fácilmente adaptables al cambio. Por otra parte, un buen diseño no debe ser específico de una aplicación concreta, sino que debe basarse en soluciones que han funcionado bien en otras ocasiones.

Si nos mantenemos dentro del ámbito del diseño de alto nivel, donde como hemos visto se encuadra la Arquitectura del Software, estaremos hablando entonces de *patrones arquitectónicos*, es decir, de esquemas de organización de los sistemas de software que determinan cuál va a ser la estructura de mismos mediante el establecimiento de su división en subsistemas, indicando las responsabilidades de cada uno de estos subsistemas y las reglas y criterios que rigen las relaciones entre ellos. Estos patrones o estilos arquitectónicos son objeto de estudio de la Arquitectura del Software y a ellos nos referiremos en la Sección 1.3.1.

Por el contrario, si nos centramos en el ámbito del diseño de bajo nivel, o diseño detallado, estaremos hablando entonces de los llamados *patrones de diseño*, a los que dedicaremos el resto de esta sección. A diferencia de los anteriores, el uso de un determinado patrón de diseño no afecta a la estructura general de una aplicación, sino sólo a una parte puntual de la misma.

Los patrones de diseño tienen su origen en los trabajos de Gamma y colaboradores [Gamma et al., 1995], quienes no sólo realizan una definición de los mismos y sugieren cómo deben describirse, sino que ofrecen todo un catálogo de patrones básicos aplicables a la programación orientada a objetos en general. Posteriormente, han sido propuestos numerosos patrones, tanto de propósito general, como específicos para multitud de áreas de aplicación.

La idea que subyace a estos patrones es la constatación de que, aunque la orientación a objetos facilita la reutilización de código, la reutilización efectiva sólo se produce a partir de un buen diseño, basado en el uso de soluciones (los patrones) que han probado su utilidad en situaciones similares. Los patrones no son bibliotecas de clases, sino más bien un esqueleto básico que cada diseñador adapta a las peculiaridades de su aplicación. Los patrones se describen en forma textual, acompañada de diagramas y pseudocódigo.

Entre las ventajas de los patrones de diseño podemos citar que son soluciones simples y técnicas que se aplican a problemas muy comunes que aparecen en el diseño orientado a objetos, facilitando la reutilización del diseño. También debemos referirnos a su carácter fundamentalmente práctico, que indica cómo resolver el problema desde el punto de vista técnico de la orientación a objetos. Entre sus inconvenientes indicaremos, en primer lugar, que son soluciones concretas a problemas concretos. Un catálogo de patrones es un libro de recetas de diseño, y aunque existen clasificaciones de patrones, todos ellos son independientes, lo que hace difícil, dado un problema determinado, averiguar si existe un patrón que lo resuelve. En segundo lugar, el uso de un patrón no se refleja claramente en el código de la aplicación. A partir de la implementación de un sistema es difícil determinar qué patrones de diseño se han utilizado en la misma. Por último, si bien permiten reutilizar el diseño, es mucho más difícil reutilizar *la implementación* del patrón. Éste describe clases que juegan papeles genéricos, explicadas normalmente con ayuda de un ejemplo, pero su implementación, adaptándolo a las peculiaridades de una aplicación en particular, consistirá en clases concretas, dependientes del dominio de aplicación. No existe un lenguaje de especificación de patrones que nos permita su reutilización por medio de la instanciación de una descripción genérica del mismo [Bosch, 1998].

1.2.3 Líneas de investigación

Dada la novedad de este campo, al menos en cuando a su consideración como objeto de estudio específico dentro de la Ingeniería del Software, existen numerosas líneas de investigación abiertas en la actualidad [Garlan, 1995]. Podríamos encuadrarlas en los siguientes grupos:

- **Modelos y lenguajes de descripción de arquitectura**. Aquí se encuadran todos los trabajos relativos a la definición de modelos arquitectónicos del software, y al diseño de notaciones específicas para la descripción de la arquitectura de los sistemas de software.
- **Estilos y patrones arquitectónicos**. Esta línea se refiere a la catalogación y clasificación de estilos arquitectónicos, a la búsqueda de patrones estructurales, a la determinación de propiedades de estilos y patrones, y a la definición de reglas que ayuden al diseñador a decidirse por un estilo u otro, en función de los requisitos concretos de su aplicación.
- **Arquitecturas de dominio específico**. Consiste en la adaptación de los estilos arquitectónicos a las características propias de un determinado dominio, e incluye todo lo relacionado con el uso marcos de trabajo composicionales y el establecimiento de familias y líneas de productos.
- **Métodos de desarrollo**. Una vez establecido qué es el diseño arquitectónico y cuáles son las tareas que implica, es necesario definir métodos de desarrollo que integren adecuadamente estos aspectos dentro de metodologías generales de desarrollo de software.
- **Herramientas y entornos de desarrollo**. Asociada a la definición de notaciones y métodos, surge la necesidad de crear herramientas y entornos de desarrollo que faciliten su uso. Se incluyen aquí editores gráficos y textuales, traductores y generadores de código, repositorios de arquitecturas y componentes, herramientas para el uso de marcos de trabajo, herramientas de análisis, etc.
- **Ingeniería inversa y reingeniería de arquitectura**. Al tratarse de una disciplina nueva, es inevitable encontrarse con numerosos sistemas en funcionamiento de los que no se ha descrito ni se conoce su arquitectura. Por tanto, es necesario el desarrollo de técnicas y herramientas de ingeniería inversa y reingeniería que permitan determinar, y rediseñar en su caso, la arquitectura de estas aplicaciones con el objeto de facilitar su mantenimiento o mejorar su eficiencia.
- **Formalismos de base**. La aplicación de métodos formales al diseño arquitectónico pasa por encontrar o desarrollar formalismos adecuados para la Arquitectura del Software. Así, es necesario el desarrollo de modelos arquitectónicos formales, establecer los fundamentos de la modularización y composición del software y realizar una caracterización formal de propiedades no funcionales tales como eficiencia, mantenibilidad, etc.
- **Técnicas de análisis**. Relacionado con los fundamentos formales de la Arquitectura del Software está el desarrollo de técnicas de análisis de propiedades estructurales basadas en los formalismos correspondientes.
- **Interoperabilidad, integración y evolución**. Se encuadra aquí todo lo relacionado con al resolución de problemas de interoperabilidad e integración entre componentes de software, condición necesaria para poder hablar de verdadera composición de componentes. También lo relacionado con el mantenimiento y evolución del diseño arquitectónico de las aplicaciones.

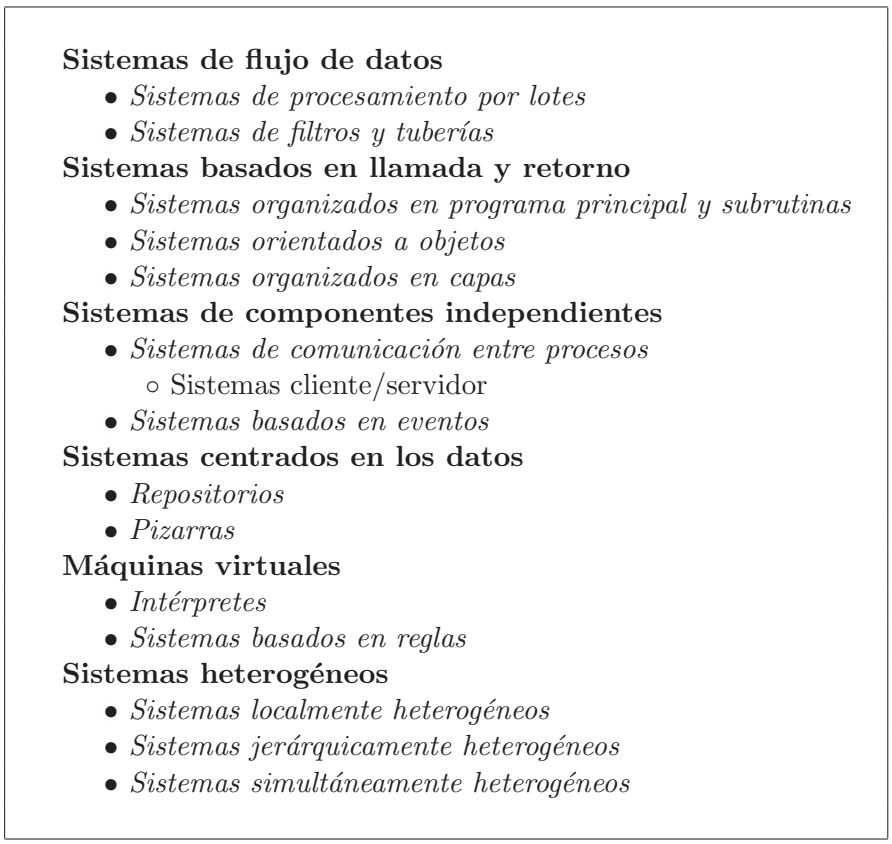


Figura 1.1: Clasificación de los principales estilos arquitectónicos.

1.3 Niveles de abstracción arquitectónicos

La Arquitectura del Software se encarga de la descripción y estudio de las propiedades estructurales de los sistemas de software. Sin embargo, este estudio puede llevarse a cabo desde diferentes niveles de abstracción. De este modo, yendo de lo más general a lo más específico, podemos distinguir varios niveles.

1.3.1 Estilos arquitectónicos

En el primero de los niveles tenemos los estilos arquitectónicos, una clasificación de los sistemas de software en grandes familias cuyos integrantes comparten un patrón estructural común. Se han catalogado numerosos estilos arquitectónicos, clasificados a su vez en variantes [Shaw y Garlan, 1996, Bass et al., 1998]. Ejemplos de estilos arquitectónicos son los sistemas de filtros y tuberías, los sistemas organizados en capas, los sistemas cliente/servidor, etc. La Figura 1.1 recoge algunos de los estilos y variantes más significativos.

A la hora de definir un estilo arquitectónico el campo de aplicación no es relevante, sólo se tiene en cuenta el patrón de organización general, los tipos de componentes presentes habitualmente en el estilo y las interacciones que se establecen entre ellos. Así, ejemplos de tipos de componentes son: clientes, servidores, filtros, niveles, bases de datos, etc., mientras que ejemplos de mecanismos de interacción son: llamadas a procedimientos o métodos, ya sean locales o remotas, paso de mensajes, protocolos de comunicación, etc.

Además de determinar los tipos de componentes y conectores involucrados, un estilo arquitectónico indicará cuáles son los patrones y restricciones de interconexión o composición entre ellos, lo que denomina como *invariantes* del estilo. Por último, asociadas a cada estilo hay una serie de propiedades que lo caracterizan, determinando sus ventajas e inconvenientes, y que condicionan la elección de uno u otro estilo para resolver un determinado problema.

Tomemos como ejemplo el estilo arquitectónico de organización en capas, representado en la Figura 1.2. En él los componentes son las capas o niveles, que pueden estar implementadas internamente tanto por objetos como por procedimientos. Cada nivel tiene asociada una funcionalidad: los niveles bajos implementan funciones simples, ligadas al hardware o al entorno, mientras que los niveles altos implementan funciones más abstractas. El mecanismo de interacción entre componentes es el de llamadas a procedimientos (o en su caso a métodos), con la restricción de que son las capas superiores las que invocan funciones o métodos de las inferiores. Una variante muy común es la que sólo permite realizar llamadas entre niveles adyacentes. Este estilo arquitectónico es el usado habitualmente para estructurar las torres de protocolos de comunicación, los sistemas operativos o incluso los compiladores.

De entre las propiedades que presenta esta organización podemos citar las siguientes:

- Se facilita la migración del sistema. El acoplamiento con el entorno está localizado en el nivel inferior. Para transportar el sistema a un entorno diferente basta con implementar de nuevo este nivel.
- Cada nivel implementa unas interfaces claras y lógicas, lo que facilita la sustitución de una implementación por otra.
- Permite trabajar en varios niveles de abstracción. Para implementar los niveles superiores no necesitamos conocer el entorno subyacente, sólo las interfaces que proporcionan los niveles inferiores.

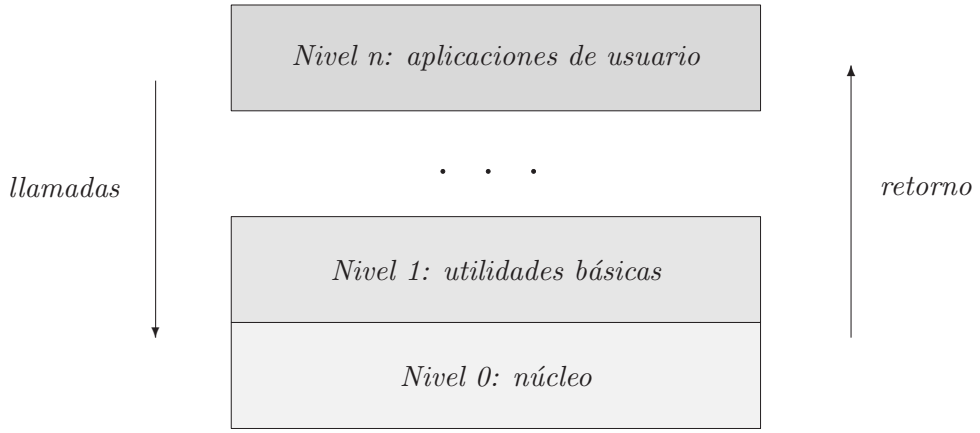


Figura 1.2: Estilo arquitectónico de organización en capas o niveles.

1.3.2 Modelos y arquitecturas de referencia

En segundo lugar están los modelos y arquitecturas de referencia, que particularizan un estilo, imponiendo una serie de restricciones sobre el mismo y realizando por lo general una descomposición y definición estándar de componentes.

Podemos encontrar numerosos ejemplos de modelos de referencia. Quizá el más conocido de todos ellos sea *Open Systems Interconnection* (OSI) [CCITT/ITU-T, 1988], recomendado por ISO. Se trata de una arquitectura de referencia que particulariza el estilo arquitectónico de organización en capas, estableciendo en siete el número de niveles y definiendo cuáles son las funciones que debe realizar cada nivel y cuál ha de ser su interfaz.

Otro ejemplo es el modelo de referencia RM-ODP para el diseño de sistemas distribuidos y abiertos [ISO/ITU-T, 1995], que tiene por objeto conseguir que la heterogeneidad de las plataformas de hardware, sistemas operativos, lenguajes, etc. sea transparente al usuario. Este modelo, basado en el estilo llamado de componentes independientes, típico de las aplicaciones distribuidas, contiene tanto elementos descriptivos como prescriptivos. Los primeros definen de forma precisa el vocabulario y conceptos que aparecen en el desarrollo de un sistema distribuido de cualquier tipo. Por su parte, los elementos prescriptivos imponen una serie de restricciones que deben cumplir los sistemas distribuidos para formar parte de este modelo. Además, se presenta una metodología de desarrollo para este tipo de sistemas, basada en la Técnica de Modelado de Objetos (OMT) [Rumbaugh et al., 1991] combinada con el uso de técnicas de descripción formal.

También debemos incluir en esta categoría los modelos de componentes surgidos recientemente en el ámbito de la Ingeniería del Software basada en Componentes, y a los que hemos hecho referencia en la Sección 1.1, tales como CCM, COM o JavaBeans. Tanto en el caso de estos últimos como en el de RM-ODP, el estilo seguido es el de sistemas basados en intercambio de eventos, particularizado en cada uno de estos modelos mediante la definición de las interfaces que los componentes que cumplan el estándar deben implementar para poder ser combinados unos con otros.

1.3.3 Marcos de trabajo

En el siguiente nivel tenemos los marcos de trabajo composicionales, de los que ya hemos hecho mención en la Sección 1.1.4, y que definen una arquitectura adaptada a las particularidades de un determinado dominio de aplicación, definiendo de forma abstracta una serie de componentes y sus interfaces, y estableciendo las reglas y mecanismos de interacción entre ellos. Normalmente se incluye además la implementación de algunos de los componentes o incluso varias implementaciones alternativas. Tal como hemos visto, la labor del usuario del marco será, por un lado, seleccionar, instanciar, extender y reutilizar los componentes que éste proporciona, y por otro, completar la arquitectura del mismo desarrollando componentes específicos, que deben ser encajados en el marco, logrando así desarrollar diferentes aplicaciones siguiendo las restricciones estructurales impuestas por el marco de trabajo.

A caballo entre el nivel de las arquitecturas de referencia y el de los marcos de trabajo está TINA [Consortium, 1995], a la que podríamos considerar como una particularización del modelo de referencia RM-ODP para el desarrollo de aplicaciones de servicios avanzados de telecomunicación. La arquitectura de servicios TINA establece una serie de principios, conceptos, mecanismos y directrices que permiten el desarrollo rápido de servicios de telecomunicación, facilitando además su gestión y operación [Chapman et al., 1995]. Para lograr estos objetivos se establece además una estructuración de este tipo de aplicaciones en tres niveles, organizados

a su vez en una serie de componentes, siendo éstos independientes del servicio concreto que se pretenda implementar. En su estado actual TINA únicamente describe los componentes que forman parte de la arquitectura propuesta, por lo que puede ser considerada como un modelo de referencia. No obstante, está en desarrollo una segunda versión que incluye la implementación de algunos de dichos componentes, lo que convertiría efectivamente a la arquitectura TINA en un marco de trabajo.

En la misma área de aplicación, podemos considerar el caso de MultiTEL [Fuentes, 1998], un marco de trabajo para el desarrollo de servicios avanzados de telecomunicación. MultiTEL está definido a partir de un modelo de composición para sistemas abiertos que sigue el estilo arquitectónico de componentes independientes, y se sustenta sobre una arquitectura genérica denominada ASTM que, al igual que TINA, establece una organización arquitectónica para servicios avanzados multimedia y define los principales componentes que se utilizan en dichas aplicaciones. Sin embargo, debemos considerar a MultiTEL como un verdadero marco de trabajo, puesto que incluye además la implementación de la arquitectura en el lenguaje Java [Fuentes y Troya, 2001]. El desarrollo de nuevos servicios se realiza por medio de la selección, extensión y especialización de los componentes proporcionados por MultiTEL, utilizando para ello herramientas visuales y lenguajes de especificación de alto nivel asociadas al marco de trabajo, y siguiendo una metodología de desarrollo diseñada a este efecto.

1.3.4 Familias y líneas de productos

Las familias o líneas de productos son un conjunto de aplicaciones similares, o bien diferentes versiones o configuraciones de la misma aplicación, de forma que todas estas aplicaciones tienen la misma arquitectura, pero cada una de ellas está adaptada o particularizada al entorno o configuración donde se va a ejecutar. Además de compartir una arquitectura, en los productos de la línea se reutilizan toda una serie de componentes que son, precisamente, los que caracterizan la línea de productos [Bosch, 2000]. Como resulta evidente, el establecimiento de familias y líneas de productos está muy ligado a una buena gestión de la configuración del software.

Las ventajas de establecer estas líneas de productos tienen que ver fundamentalmente con la reutilización, ya que implican una disminución de costes y un aumento de la calidad y fiabilidad del producto. Estas ventajas se materializan en:

- **Reutilización de la arquitectura.** Todas las aplicaciones de la línea comparten básicamente la misma arquitectura, lo que permite reutilizar su diseño, su documentación, asegurar el cumplimiento de determinadas propiedades, etc.
- **Reutilización de los componentes.** Como ya hemos indicado, parte de los componentes son comunes a los distintos productos, o bien pueden obtenerse parametrizando componentes genéricos. Esto facilita notablemente la implementación de nuevos integrantes de la línea de productos.
- **Fiabilidad.** Los componentes empleados están exhaustivamente probados, al ser compartidos con aplicaciones que ya están en funcionamiento, lo que permite corregir defectos y aumentar su fiabilidad y calidad. Por otro lado, el uso de dichos componentes en aplicaciones que comparten la misma arquitectura permite garantizar que no hay pérdida de fiabilidad en la reutilización.
- **Planificación.** La experiencia repetida en el desarrollo de productos de la línea permite hacer una planificación más fiable del proyecto de desarrollo, y con mayores posibilidades de ser cumplida.

- **Mejora del proceso.** También es posible adaptar y especializar los propios procesos de desarrollo para la línea de productos.

1.3.5 Instancias arquitectónicas

En último lugar, tenemos las instancias arquitectónicas, que representan la arquitectura de un sistema de software concreto, caracterizada por los módulos o componentes que lo forman y las relaciones que se establecen entre ellos. Como es lógico, cualquier aplicación tiene una arquitectura que puede ser descrita y estudiada y que presenta determinadas propiedades, ya sea o no la aplicación parte de una familia o línea de productos, haya sido o no desarrollada utilizando un marco de trabajo, siga o no un modelo o arquitectura de referencia, o incluso pueda ser catalogada dentro de un estilo arquitectónico o siga un estilo heterogéneo.

1.4 Descripción de la arquitectura

Como ya hemos indicado, la Arquitectura del Software tiene como objetivo el conocimiento, análisis y reutilización de la arquitectura de los sistemas de software. Para ello, es necesario hacer explícita dicha arquitectura, lo que implica el uso de algún lenguaje.

Podemos considerar diversas opciones a la hora de elegir una notación para representar la arquitectura de un sistema de software: notaciones diagramáticas o textuales, específicas o utilizadas para describir otros aspectos del software, formales o informales. En esta sección daremos un repaso a algunas de estas notaciones.

1.4.1 Notaciones diagramáticas

La forma más intuitiva y que, por tanto, ha venido utilizándose tradicionalmente de representar la arquitectura de un sistema de software consiste en el uso de diagramas informales de cajas y líneas, donde las cajas representan componentes y las líneas conectores o mecanismos de interacción. Como ejemplo, la Figura 1.3 muestra la descripción, utilizando un diagrama de este tipo, de un *Entorno de Validación de Protocolos (EVP)* [GISUM, 1995, Merino y Troya, 1996] desarrollado por nuestro grupo de investigación como parte del Plan Nacional de Banda Ancha (PlanBa). En el *EVP* los protocolos de comunicación se especifican haciendo uso de alguna técnica de descripción formal (TDF), siendo a continuación traducidos a un lenguaje lógico concurrente (LCC) para su validación o simulación sobre un núcleo distribuido.

La leyenda que acompaña al diagrama de la Figura 1.3 intenta aclarar el significado de los diversos tipos de componentes utilizados en el mismo, lo que permite distinguir unos de otros y hace que el diagrama aporte mayor información. Sin embargo, no sucede esto con los mecanismos de interacción, y así las líneas en diagramas de este tipo pueden significar flujo de datos o de control, invocaciones explícitas de procedimientos o métodos, paso de mensajes, o incluso el uso de complejos protocolos de comunicación entre componentes. Las limitaciones de utilizar estos diagramas informales son evidentes:

- Carecen de una semántica definida, por lo que estas descripciones son imprecisas y pueden ser interpretadas de forma distinta por personas diferentes. En cada diagrama cambia el significado de los elementos que aparecen en él.
- La interpretación ni siquiera es consistente dentro del mismo diagrama: el mismo elemento gráfico puede utilizarse para hacer referencia a conceptos distintos.

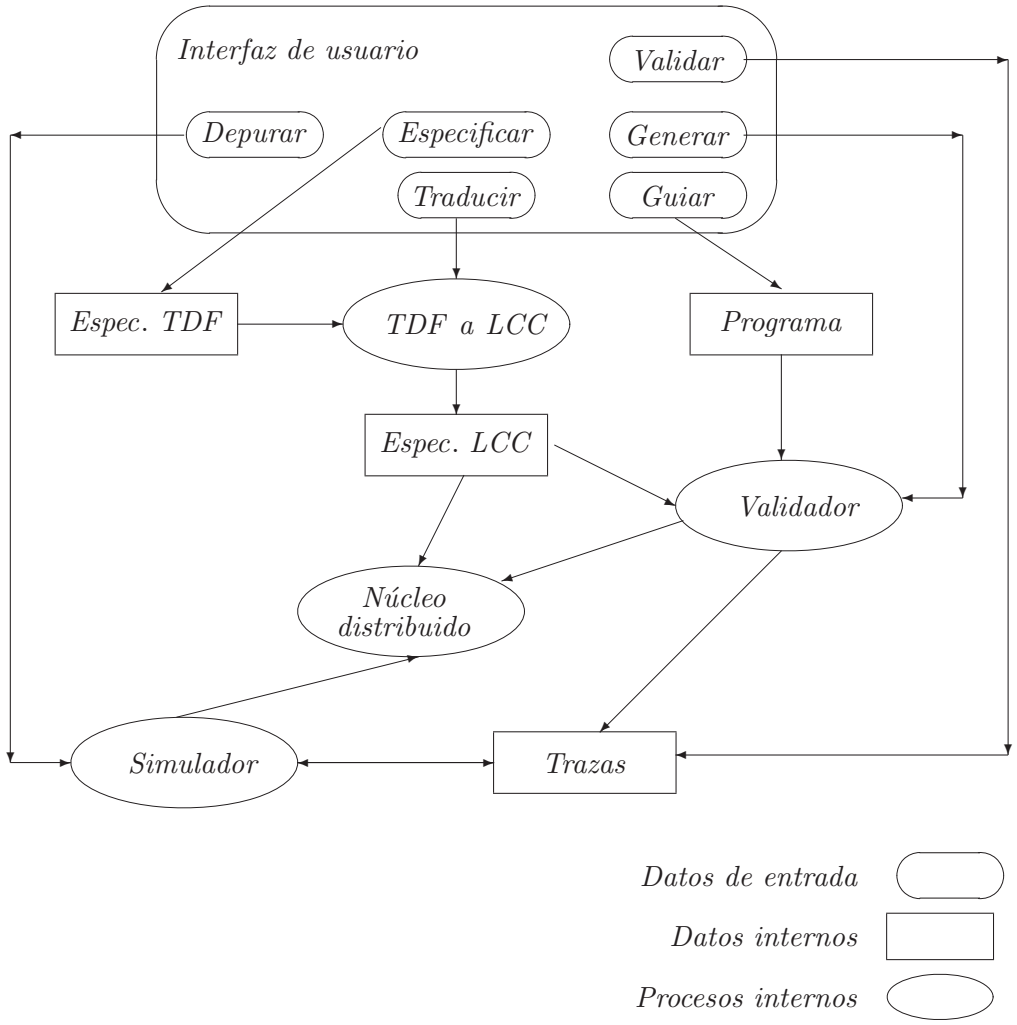


Figura 1.3: Descripción informal de un Entorno de Validación de Protocolos.

- Esta imprecisión y ambigüedad hace que no puedan ser objeto de análisis, con lo que no es posible predecir las propiedades de los sistemas que representan.

- La relación entre la arquitectura que describe el diagrama y la del sistema realmente implementado es, por lo general, tenue. Como consecuencia del desfase entre la documentación y la implementación, el mantenimiento se hace más difícil, y se complica también la reutilización de la arquitectura desarrollada.

Existen, no obstante, notaciones diagramáticas que están definidas de forma más precisa, con lo cual se solventan en parte los problemas citados. Nos estamos refiriendo a las notaciones asociadas a los métodos de desarrollo de software, los cuales definen habitualmente diagramas para representar la estructura de alto nivel de las aplicaciones, como, por ejemplo, los diagramas de arquitectura utilizados en algunos métodos estructurados [Hatley y Pirbhai, 1987]. Como muestra representativa de esta categoría, consideraremos el Lenguaje Unificado de Modelado (UML), al que dedicamos la sección siguiente.

1.4.2 El Lenguaje Unificado de Modelado

El Lenguaje Unificado de Modelado (UML) es una notación para especificar, visualizar y documentar sistemas de software desde la perspectiva de la orientación a objetos. Su objetivo es representar el conocimiento acerca de los sistemas que se pretenden construir y las decisiones tomadas durante su desarrollo, tanto lo referido a su estructura estática como a su comportamiento dinámico. UML postula un proceso de desarrollo iterativo, incremental, guiado por los casos de uso y *centrado en la arquitectura* [Booch et al., 1999]. La representación en UML de un sistema de software consta de cinco vistas o modelos parciales separados, aunque relacionados entre sí, denominadas vista de casos de uso, de diseño, de implementación, de procesos y de despliegue. Cada uno de estos modelos representa el sistema por medio de diversos diagramas. Aunque no existe de forma explícita una vista arquitectónica, estas cinco vistas pretenden describir, en su conjunto, la arquitectura del sistema [Kruchten, 1995].

En favor de UML podemos señalar que es un lenguaje gráfico con sintaxis y semántica bastante bien definidas. La sintaxis de la notación gráfica se especifica mediante su correspondencia con los elementos del modelo semántico subyacente [Rumbaugh et al., 1999], cuya semántica se define de manera semi-formal por medio de un metamodelo, textos descriptivos y restricciones. Existen además numerosas iniciativas para dotar a esta notación de una semántica formal [Breu et al., 1997, Evans y Kent, 1999, Whittle, 2000, Toval y Fernández-Alemán, 2001].

Por otra parte, el lenguaje es extensible, de forma que pueden añadirse nuevas construcciones para abordar aspectos del desarrollo de software no previstos inicialmente en la notación. Esta extensión puede realizarse bien mediante la especialización de los conceptos del metamodelo (lo que hace que la notación extendida ya no sea compatible con las herramientas existentes), o bien mediante la definición de restricciones, valores etiquetados y *estereotipos*, sin modificar la sintaxis ni la semántica de UML [Toval y Fernández-Alemán, 2000].

En cuanto a su capacidad para describir la arquitectura, debemos señalar que UML maneja conceptos utilizados también por la Arquitectura del Software, como son los de interfaz, componente o conexión, y que los mecanismos de extensión disponibles pueden utilizarse para definir otros conceptos no contemplados o para establecer restricciones que definan de forma más precisa la semántica de estos conceptos.

No obstante, debemos de partir de la base de que el propósito primordial de un lenguaje es proporcionar un vehículo de expresión para las nociones intuitivas y prácticas de sus usuarios [Shaw y Garlan, 1995], en este caso, los arquitectos del software. Si ciertas abstracciones fundamentales utilizadas por los arquitectos (por ejemplo, los componentes o los conectores) quedan ocultas o desvirtuadas debido a su representación mediante las abstracciones que proporciona el lenguaje (en este caso las clases), la labor del arquitecto se ve dificultada de forma innecesaria [Medvidovic y Rosenblum, 1999]. Podemos concluir, por tanto, que a pesar de las afirmaciones de sus autores, el uso previsto de UML no es la descripción de la arquitectura del software como tal, de forma que las expresividad para modelar sistemas ofrecida por UML no satisface de manera completa las necesidades de la descripción arquitectónica. En particular, resulta deficiente para la descripción de los aspectos dinámicos de las estructuras, los protocolos de comunicación entre componentes y la correspondencia entre las distintas vistas de la arquitectura [Hofmeister et al., 1999].

Una alternativa al uso de notaciones diagramáticas consiste en utilizar algún lenguaje que sea adecuado para la representación arquitectónica. Con ello, conseguiremos cuando menos, una semántica más precisa. En este sentido, podemos pensar en utilizar un lenguaje de programación modular, o bien un lenguaje de interconexión de módulos o de descripción de interfaces. De ellos nos ocuparemos en la próximas secciones.

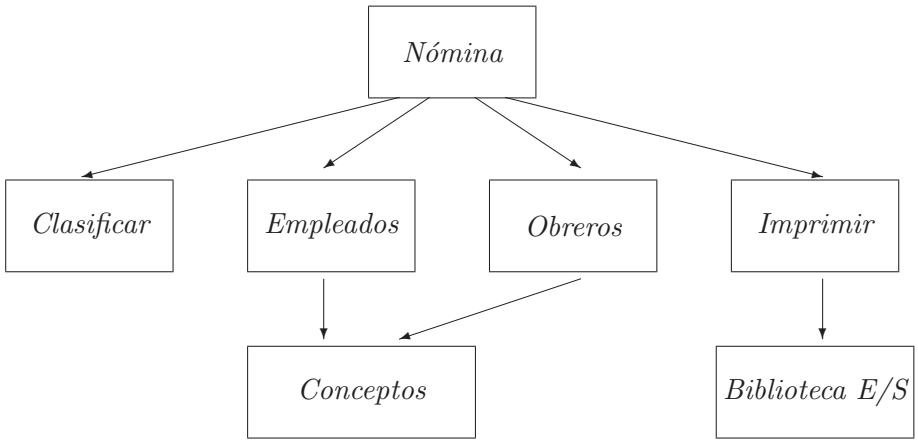


Figura 1.4: Descripción de dependencias modulares en un sistema de nómina.

1.4.3 Lenguajes de programación modular

Los lenguajes de programación disponen de una serie de estructuras sintácticas que permiten modularizar los sistemas de software implementados con ellos. De este modo, podemos representar los componentes del sistema mediante módulos del lenguaje, y describir sus interfaces y relaciones mediante las relaciones de exportación e importación entre módulos.

El inconveniente que presenta esta aproximación es que, al igual que sucede con UML, la intención de estos mecanismos de modularización no es la de describir los componentes del sistema y sus relaciones durante su ejecución, sino que viene dada por las necesidades de modularizar el código fuente, tanto para mayor claridad y más fácil mantenimiento, como para permitir la compilación de cada módulo de forma independiente. Por tanto, con esta aproximación se mezclan y confunden aspectos de programación con aspectos estructurales.

Por otro lado, las interfaces basadas en el uso de cláusulas de importación y exportación no indican nada acerca de los protocolos de interacción entre los módulos, es decir, sobre cuál es el orden que rige la interacción entre los componentes. La semántica que ofrecen las estructuras de modularización de los lenguajes de programación está establecida de antemano, y se refiere a mecanismos de coordinación y comunicación prefijados, tales como la invocación de procedimientos o métodos, o la compartición de datos.

Para arrojar luz sobre los diferentes puntos de vista que ofrece una especificación arquitectónica y la descripción de un diseño de descomposición modular, consideremos por ejemplo un sistema sencillo —una nómina— que procesa un flujo de datos correspondientes a los trabajadores de una empresa. Según sean éstos obreros o empleados, el cálculo de la nómina se realiza de forma diferente, aunque parte de los conceptos de la nómina son comunes a ambos tipos de trabajadores. Supongamos que este sistema ha sido diseñado siguiendo el estilo de filtros y tuberías, de forma que el flujo de datos de trabajadores se divide (usando un filtro *Clasificar*) de acuerdo con el tipo de empleado, y cada flujo resultante se procesa de forma separada (por medio de sendos filtros *Empleados* y *Obreros*), para finalmente reunir de nuevo ambos flujos e imprimir los recibos de nómina (usando para ello un filtro *Imprimir*). En una implementación típica de este sistema nos encontraríamos con un diseño modular como el que muestra la Figura 1.4, que consiste en un módulo principal (*Nómina*), un módulo para el tratamiento de los conceptos comunes (*Conceptos*), un módulo para operaciones de entrada y

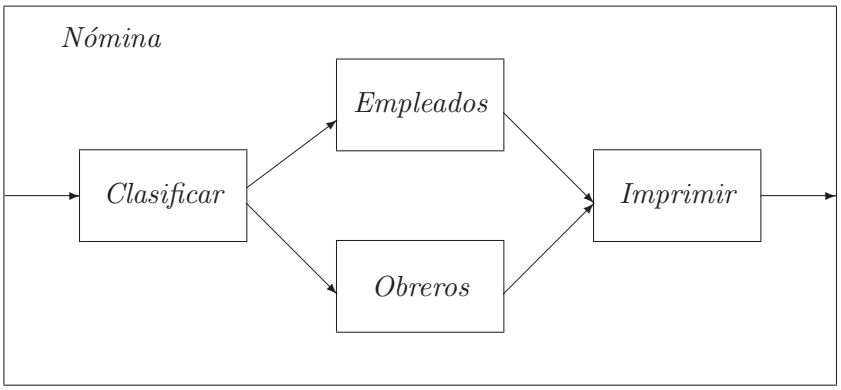


Figura 1.5: Descripción arquitectónica un sistema de nómina.

salida y, finalmente, los módulos ya citados para llevar a cabo las transformaciones deseadas. El módulo principal depende del resto, puesto que debe coordinar todas las transformaciones y realizar las conexiones necesarias de los flujos de datos.

Aunque útil, este diagrama de descomposición modular no es capaz de mostrarnos el sistema desde un punto de vista arquitectónico. El diagrama indica qué módulos están presentes en el sistema de nómina, así como las dependencias modulares que permiten compilar y ensamblar los módulos de código fuente. Sin embargo, no da una visión de la arquitectura del sistema, que como hemos dicho queremos que siga el estilo de filtros y tuberías, ni muestra tampoco las conexiones entre filtros, ni el flujo de información a través del sistema.

La Figura 1.5 ofrece una representación alternativa del sistema. En contraste con el diagrama anterior, las relaciones entre componentes (es decir, las tuberías) aparecen ahora de forma explícita en el diagrama, donde se representan mediante flechas, mientras que ya no se muestran las dependencias modulares. Esta segunda descripción realza claramente el diseño arquitectónico y demuestra cómo, con objeto de comprender la estructura de alto nivel de un sistema de software, es importante expresar no sólo las dependencias de definición y uso entre módulos de implementación, sino también las interacciones abstractas que resultan de la composición final que se hace de los componentes. Por supuesto, para que el diagrama tenga significado debe existir un acuerdo a la hora de interpretar las cajas y líneas como filtros y tuberías, respectivamente.

Por último, se podría pensar en utilizar un lenguaje que permitiese formas alternativas de interacción entre componentes, con lo que lograríamos una mayor expresividad a la hora de describir las interacciones entre ellos. Por ejemplo, el lenguaje de *shell* de Unix permite utilizar tuberías para conectar filtros, con lo que resulta sencillo describir la estructura de sistemas que sigan este estilo arquitectónico. Por otro lado, el lenguaje Ada incorpora un mecanismo de citas para la sincronización de procesos distinto al de invocación explícita de métodos comúnmente usado en los lenguajes de programación modular. Por último, existe toda una serie de lenguajes y sistemas especializados que permiten describir la arquitectura de sistemas que siguen un determinado estilo [Magee et al., 1989, Kramer, 1990, Mak, 1992, Beach, 1992, Medvidovic et al., 1996, SEI, 1990]. En cualquier caso, el inconveniente asociado a todas estas aproximaciones viene dado por su falta de flexibilidad: los mecanismos de interacción siguen siendo limitados y establecidos de antemano.

1.4.4 Lenguajes de interconexión de módulos y de descripción de interfaz

Los lenguajes de interconexión de módulos (MILs) están más próximos a las necesidades de expresividad de la Arquitectura del Software, con lo que se resuelven algunos de los problemas citados anteriormente. En primer lugar, permiten establecer una separación más clara entre la configuración del sistema, es decir, los aspectos de estructuración del sistema e interconexión de componentes, y la descripción de las partes o módulos del mismo [Prieto-Díaz y Neighbors, 1986]. Así mismo, algunos de estos lenguajes disponen de mecanismos de parametrización para describir plantillas para la composición de sistemas. Un ejemplo de este tipo de lenguajes es Standard ML (SML) [Milner et al., 1990].

Sin embargo, los MILs están orientados fundamentalmente a la interconexión de interfaces de importación y exportación de unidades de programación de bajo nivel, resolviendo por ejemplo cuestiones como el renombramiento de operaciones, pero presentan básicamente las mismas deficiencias que los anteriores: las únicas relaciones que describen son las de dependencia del código fuente, vía cláusulas de importación y exportación, e implican un único mecanismo de interacción, la llamada a procedimiento.

Por su parte, los lenguajes de descripción de interfaz (IDLs) pueden ser considerados como una generación posterior a los MILs. Estos lenguajes se utilizan para declarar la interfaz sintáctica de un componente, facilitando toda la información necesaria para usar dicho componente a otros que actuarán como clientes del mismo. Esta información incluye los atributos y operaciones que pertenecen a la interfaz, así como los parámetros de cada operación y las excepciones que puedan producirse. De esta forma, la interfaz constituye un contrato entre un cliente y un proveedor y presenta la flexibilidad y el encapsulamiento necesarios para que los componentes hagan uso de los servicios que ofrecen unos a otros.

La notación usada para describir la interfaz es independiente del lenguaje de programación utilizado para implementar los componentes, y la semántica del IDL define la forma en la que los tipos y funciones del IDL se transforman en los tipos y funciones de cada uno de los lenguajes de programación considerados. A partir de la descripción de la interfaz del componente, los compiladores de IDL son los encargados de generar un conjunto de módulos en el lenguaje de programación escogido, de forma que es posible la interconexión de componentes implementados en lenguajes heterogéneos.

Sin duda, de entre los lenguajes de descripción de interfaz, el más significativo es el IDL de la plataforma de objetos distribuidos de CORBA [OMG, 1999]. Este lenguaje, denominado CORBA IDL, permite la descripción de la interfaz que exporta un objeto en una notación textual similar a C++ y que incorpora también los conceptos de herencia y polimorfismo. Estas interfaces constan de declaraciones de tipos (formados a partir de los tipos básicos contemplados en el IDL), y de los atributos públicos y signatura (parámetros, valor de retorno y excepciones) de las operaciones exportadas por un objeto. En la Figura 1.6 se muestra la descripción de la interfaz de un objeto *CuentaCorriente* utilizando esta notación.

Como se puede apreciar, el objeto *CuentaCorriente* ofrece a sus clientes una interfaz formada por dos atributos: *titular* (que se declara como no modificable) y *saldo*, y dos métodos: *ingreso* y *reintegro*. Adicionalmente, se detallan las posibles excepciones (en este caso *saldoInsuficiente*) que pueden elevarse como consecuencia de la invocación de dichos métodos.

Esta información basta para que puedan establecerse los mecanismos para registrar, localizar y dirigir peticiones de servicios al objeto *CuentaCorriente*, pero no indica nada acerca de la arquitectura de las aplicaciones donde aparezca el componente, ni tampoco de los protocolos de interacción que siguen los objetos que forman el sistema. El único mecanismo de interacción

```
interface CuentaCorriente {
   readonly attribute string titular;
   attribute float saldo;
   exception saldoInsuficiente { float saldo;};

   void ingreso(in int importe, out string recibo);
   void reintegro(in int importe, out string recibo);
      raises(saldoInsuficiente);
   };
```

Figura 1.6: Especificación de la interfaz de un objeto *CuentaCorriente* en CORBA IDL.

considerado es la llamada (remota) a procedimientos, y de hecho, en CORBA IDL sólo se indican cuáles son los métodos que un componente exporta, pero no los que precisa para su funcionamiento, con lo que no se hacen explícitas las dependencias de contexto del mismo.

Recientemente se han desarrollado diversos trabajos que proponen distintas extensiones de CORBA IDL para incluir información más detallada acerca de cómo utilizar o combinar los componentes CORBA. Estos trabajos postulan el uso de reglas lógicas y temporales [Lea, 1995], redes de Petri [Bastide et al., 1999], o álgebras de procesos [Canal et al., 2000].

1.4.5 Lenguajes de descripción de arquitectura

Según lo anteriormente expuesto, resulta evidente la necesidad de una notación específicamente desarrollada para la descripción de la arquitectura del software, de forma que permita abordar de manera adecuada los problemas y necesidades de este nivel de diseño. Estos son los denominados lenguajes de descripción de arquitectura (ADLs, de sus siglas en inglés). Este tipo de lenguajes son necesarios para disponer de abstracciones útiles para modelar sistemas complejos desde un punto de vista arquitectónico, a la vez que deben permitir un nivel de detalle suficiente como para describir propiedades de interés de dichos sistemas. De esta manera será posible comprobar, ya desde las etapas iniciales del desarrollo de un sistema, si éste cumple o no determinados requisitos.

No existe aún un consenso claro dentro de la comunidad científica respecto a lo que es un ADL, ni sobre qué aspectos de los sistemas de software deben ser contemplados por este tipo de lenguajes [Medvidovic y Taylor, 2000, Collins-Cope, 2000]. No obstante, entre las características que de estos lenguajes podemos considerar las siguientes [Shaw y Garlan, 1996]:

- **Composición.** Permiten la representación del sistema como composición de una serie de partes.
- **Configuración.** La descripción de la arquitectura es independiente de la de los componentes que formen parte del sistema.
- **Abstracción.** Describen los *roles* o papeles abstractos que juegan los componentes dentro de la arquitectura.
- **Flexibilidad.** Permiten la definición de nuevas formas de interacción entre componentes.

- **Reutilización.** Permiten la reutilización tanto de los componentes como de la propia arquitectura.
- **Heterogeneidad.** Permiten combinar descripciones heterogéneas.
- **Análisis.** Permiten diversas formas de análisis de la arquitectura y de los sistemas desarrollados a partir de ella.

De un modo general, podríamos decir que un ADL se centra en la estructura de alto nivel de un sistema de software, más que en los detalles de implementación de los módulos de código fuente que lo constituyen. Los ADLs proporcionan tanto una sintaxis específica como un marco conceptual para modelar la arquitectura de un sistema de software [Vestal, 1993]. Los conceptos fundamentales manejados en una descripción arquitectónica son los siguientes:

- **Componentes.** Representan unidades de computación o de almacenamiento de datos.
- **Conectores.** Son utilizados para modelar las interacciones entre componentes y las reglas que gobiernan dichas interacciones.
- **Configuraciones arquitectónicas.** Grafos de componentes y conectores que describen la estructura arquitectónica de los sistemas de software.

El objeto no es describir los detalles internos de los componentes del sistema, sino sólo su interfaz, es decir la información necesaria para interconectar dichos componentes y formar así sistemas mayores. Como veremos más adelante, esta descripción de las interfaces no se limita —tal como sucede en los lenguajes de programación convencionales— a la signatura de los métodos que ofrece o requiere el componente, sino que incluye información sobre la funcionalidad del componente, los patrones de interacción que utiliza en su funcionamiento, y otras características diversas. Si además de esto se pretende inferir algún tipo de propiedad a partir de una descripción arquitectónica, las interfaces deben estar modeladas formalmente.

De acuerdo con esto, el marco conceptual de un ADL suele incluir una teoría formal, en términos de la cual se expresa la semántica del lenguaje. El marco formal elegido determina la adecuación del ADL para modelar determinados tipos de sistemas (por ejemplo, sistemas altamente concurrentes, sistemas organizados en capas, etc.) o para modelar determinados aspectos de un sistema (por ejemplo, propiedades de viveza, propiedades de tiempo real, etc.)

A pesar de la relativa juventud de este campo, existen diversos ejemplos de ADLs. En estos lenguajes se separa claramente lo que es la descripción de la arquitectura o configuración del sistema de la descripción e implementación de los componentes. Además, muchos de ellos tienen una base formal (por lo general un álgebra de procesos), lo que permite el análisis y verificación de propiedades de los sistemas descritos. No obstante, el mayor inconveniente que presentan es que estamos tratando con un campo relativamente reciente y por tanto muy inmaduro e inestable. La difusión de estos lenguajes es aún reducida.

A continuación se describen brevemente las características principales de algunos de los ADLs más significativos. Estudios comparativos más detallados pueden encontrarse en [Medvidovic y Taylor, 2000, Medvidovic y Rosenblum, 1997].

Unicon

UniCon (*Language for Universal Connector Support*), desarrollado por Shaw y colaboradores [Shaw et al., 1995], establece un modelo para especificar la arquitectura del software ha-

Componentes	**Atributos y actores**
Module	*Library, EntryPoints, Processor, ...* *RoutineDef, RoutineUse, GlobalDataDef, GlobalDataUse,* *ReadFile, WriteFile, ...*
SharedData	*Library, Processor, ...* *GlobalDataDef, GlobalDataUse, ...*
SeqFile	*Library, RecordFormat, ...* *ReadNext, WriteNext*
Filter	*Processor* *StreamIn, StreamOut*
SchedProcess	*Processor, Priority, TriggerDef, ...* *RPCDef, RPCCall, RTLoad*
Conectores	**Atributos y roles**
Pipe	*PipeType (Unix), MaxConns* *Source, Sink*
FileIO	*Reader, Readee, Writer, Writee*
RemoteProcCall	*Definer, Caller*
DataAccess	*Definer, User*
RTScheduler	*Algorithm, Processor, Trace, ...* *Load*

Figura 1.7: Algunos de los componentes y conectores predefinidos en UniCon, con sus atributos y papeles más relevantes.

ciendo uso de abstracciones predefinidas para describir los componentes, las interacciones entre dichos componentes (caracterizadas en este lenguaje como conectores) y los patrones que dirigen la composición de unos con otros para formar sistemas. El objetivo del lenguaje es servir de vehículo de representación de las abstracciones utilizadas en la práctica por los diseñadores de software, para lo cual los mecanismos de conexión ofrecidos por el lenguaje deben ser variados. Además, se pretende establecer un vínculo entre especificación arquitectónica realizada en UniCon y la implementación final del sistema en un lenguaje de programación convencional.

Desde el punto de vista de UniCon, los sistemas de software están formados por dos categorías diferenciadas de elementos: los componentes y los conectores. Los componentes se corresponden básicamente con unidades de compilación de los lenguajes de programación convencionales y con otros objetos del nivel del usuario, como por ejemplo un fichero. Es en ellos donde residen tanto las computaciones como la memoria del estado del sistema. Por su parte, los conectores tienen un carácter menos tangible, no pudiendo identificarse tan fácilmente en el nivel del usuario. Ejemplos de conectores en UniCon son: entradas en una tabla, directivas de ensamblado, estructuras de datos utilizadas en tiempo de ejecución, secuencias de llamadas a procedimientos o al sistema encastradas en un programa, protocolos estándares de comunicación, etc.

En UniCon tanto los componentes como los conectores tienen asociado un tipo o clase, que determina cuál va a ser su interfaz, indicando el papel que el elemento juega en un sistema. Cada tipo (de componentes o conectores) está descrito mediante una serie de atributos y tiene también

```
connector Unix-Pipe
    protocol is
        type Pipe
        role source is Source
            MaxConns (1)
        end source
        role sink is Sink
            MaxConns (1)
        end sink
    end protocol
    implementation is
        builtin
    end implementation
end Unix-Pipe
```

Figura 1.8: Especificación del conector *Unix-Pipe* en UniCon.

asociada una implementación determinada. En el caso de los componentes, estos atributos describen características tales como su funcionalidad, invariantes, rendimiento, etc, mientras que su implementación suele consistir en código en un lenguaje de programación convencional. En cambio, la especificación de los atributos de un conector puede incluir características tan diversas como la garantía de entrega de paquetes en una red de comunicación, restricciones sobre el orden de envío o recepción de eventos, reglas sobre la instanciación de parámetros, restricciones sobre el número de componentes que conectan y los papeles que estos componentes juegan en la conexión, etc. La implementación de los conectores es así mismo diversa: por medio de mecanismos proporcionados por los lenguajes de programación (como por ejemplo variables globales o llamadas a procedimiento), por medio de entradas en tablas de tareas o de encaminamiento, mediante llamadas a funciones del sistema operativo o una plataforma subyacente (como por ejemplo para la lectura y escritura de *sockets* o para el envío y recepción de mensajes), etc.

Por su parte, la descripción de la interfaz se utiliza para definir las posibilidades de conexión entre componentes y conectores para formar sistemas mayores. En el caso de los componentes, esta interfaz está formada por un conjunto de *actores*, mientras que en el caso de los conectores viene descrita por medio de una serie de *roles*. Tanto actores como roles tienen asociado, a su vez, un tipo y se describen mediante una serie de atributos que dependen del tipo de que se trate. La Figura 1.7 muestra algunos de los tipos de componentes y conectores existentes en UniCon junto con los atributos y papeles (actores o roles) que los caracterizan, mientras que la Figura 1.8 muestra la especificación en este lenguaje de un conector tubería como los existentes en Unix. En esta última observamos cómo la interfaz de la tubería consta de dos roles (denominados *source* y *sink*) de los tipos predefinidos *Source* y *Sink*, respectivamente. En ambos casos, el número máximo de conexiones (*MaxConns*) que admiten dichos roles es uno, lo que caracteriza estas tuberías como las típicas del sistema operativo Unix. Respecto a la implementación de las tuberías, vemos que está predefinida (*built in*), es decir, que es conocida por el propio lenguaje.

A partir de estos componentes y conectores simples, la especificación de un subsistema o componente compuesto en UniCon consistirá en la descripción de una serie de conexiones entre los actores de los componentes y los roles de los conectores. Los actores y roles conectados deben ser compatibles entre sí. Así por ejemplo, para construir un cauce, formado por una sucesión de filtros interconectados por medio de tuberías, conectaremos los actores *StreamOut* y *StreamIn* de dos componentes filtro (*Filter*) consecutivos a través de los roles *Source* y *Sink*, respectivamente, de un conector tubería (*Pipe*).

Como ya hemos apuntado, uno de los objetivos de UniCon es el establecimiento de un vínculo entre la arquitectura de los sistemas de software y su implementación. Por este motivo, el lenguaje está acompañado de un generador de código capaz de construir un sistema ejecutable a partir de su especificación arquitectónica. Esta posibilidad de obtener de forma automática el sistema final es la característica más relevante de UniCon, pero es precisamente lo que hace que tanto su sintaxis como su estructura dependan en gran medida de las particularidades de los distintos tipos de componentes y conectores contemplados. Tal como se puede apreciar en la Figura 1.7, existe una gran variabilidad entre los distintos elementos que maneja el lenguaje, y entre los atributos específicos de cada uno de ellos y de los actores o roles que representan su interfaz.

Esta diversidad provoca que en UniCon los tipos que representan a estas abstracciones estén predeterminados y formen parte del propio lenguaje (es decir, sean tipos básicos del mismo), estando componentes y conectores descritos por medio de una compleja notación *ad hoc*, difícil de entender y manejar. Este control estricto sobre los tipos componentes y conectores que pueden utilizarse y cuáles son las posibilidades de combinarlos es precisamente lo que permite a UniCon la generación de un sistema ejecutable a partir de la especificación arquitectónica. Para ello basta con asociar cada componente del sistema con el elemento de software que lo implementa (ya sea un módulo binario, un fichero, etc.) y conectar estos componentes mediante conectores de los tipos previstos en la arquitectura, utilizando para ello los mecanismos específicos para cada tipo de conector, y que son conocidos por la herramienta de generación de código.

Para finalizar, debemos señalar que este lenguaje carece de una base formal que permita la validación de propiedades de las especificaciones. Tampoco se contemplan en él mecanismos para la reutilización de componentes o arquitectura, ni para su adaptación a cambios en los requisitos, ni es posible la descripción de sistemas cuya estructura vaya evolucionando durante su ejecución.

Wright

Un ADL similar a UniCon, en cuanto a su estructura y a los conceptos que maneja, es Wright [Allen y Garlan, 1997, Allen et al., 1998]. En efecto, este lenguaje distingue también entre componentes y conectores y dispone de estructuras sintácticas diferenciadas para describir la interfaz de unos y otros. Así, mientras la interfaz de un componente está representada en Wright por medio de una serie de *puertos* (equivalentes por tanto a los actores de UniCon), la de un conector se describe también en este lenguaje por medio de una serie de *roles*. Cabe señalar que estos elementos de descripción de interfaz —puertos y roles— no son equivalentes, puesto que presentan una clara asimetría: los puertos describen el comportamiento *efectivo* de los componentes, es decir, describen el comportamiento de los componentes tal cual es, mientras que los roles describen el comportamiento *esperado* por los conectores, es decir, el comportamiento que deben tener los componentes que conectemos a ellos.

```
connector Pipe =
    role Writer = write → Writer ⊓ close → √
    role Reader = let ExitOnly = close → √
                  in let DoRead = (read → Reader □ readEof → ExitOnly)
                     in DoRead ⊓ ExitOnly
    glue = let ReadOnly = Reader.read → ReadOnly
                        □ Reader.readEof → Reader.close → √
                        □ Reader.close → √
           in let WriteOnly = Writer.write → WriteOnly □ Writer.close → √
              in Writer.write → glue □ Reader.read → glue
                 □ Writer.close → ReadOnly □ Reader.close → WriteOnly
```

Figura 1.9: Especificación del conector *Pipe* en Wright.

También de forma análoga a como ocurre en UniCon, en Wright los sistemas se construyen a partir de la conexión de puertos de componentes y roles de conectores que presenten un comportamiento compatible. Sin embargo, terminan con esto los paralelismos entre ambos lenguajes, dado que el principio que inspira el diseño de Wright no es la obtención de forma automática de un sistema ejecutable (lo que, como hemos visto, lleva en UniCon a la inclusión en el lenguaje de toda una colección de tipos predefinidos), sino que el objetivo consiste aquí en el análisis formal de los sistemas descritos.

Por esta razón, no existen tipos de componentes y conectores predefinidos en el lenguaje, y las interfaces, en lugar de consistir en la parametrización de una serie de atributos específicos, indican cuál es el *protocolo* que rige el comportamiento del componente o conector que se está especificando. Para esta descripción de comportamiento se utiliza el álgebra de procesos CSP [Hoare, 1985]. Por medio de este formalismo se describe la interacción (es decir, el protocolo de coordinación) que tiene lugar entre el elemento y su entorno, mediante la sincronización de eventos que sirven para modelar tanto la invocación de operaciones, como el intercambio de mensajes.

Para ilustrar estos conceptos, en la Figura 1.9 se muestra la especificación en Wright de un conector tubería (*Pipe*), equivalente al especificado en UniCon en la Figura 1.8. Como se puede apreciar, la interfaz de una tubería está formada por dos roles, denominados *Writer* y *Reader*, que se encargan de describir los papeles —de escritor y de lector— que los componentes que vayan a ser conectados con la tubería juegan respecto a ella.

Así, vemos que el componente que actúe como escritor se puede sincronizar con la tubería mediante dos eventos alternativos, uno de lectura y otro de cierre de la tubería (representados como *write* y *close*, respectivamente). Si la sincronización se produce por medio de un evento *write*, el rol vuelve a su situación original (*Writer*), mientras que si la sincronización se produce con un evento *close*, el rol finaliza con éxito, lo que en CSP se indica mediante $\surd$ (véase la Sección 1.5.1 para más información sobre este álgebra de procesos). La elección entre una y otra alternativa se realiza de forma local (lo que viene indicado por el operador $\sqcap$), de forma que corresponde al componente conectado a este rol (es decir, a quien escribe en la tubería) la decisión de finalizar las operaciones de escritura.

Por otro lado, el rol *Reader*, correspondiente al comportamiento que se espera del componente que actúe como escritor, determina que la tubería es capaz de sincronizarse en un número indeterminado de eventos de lectura *read* que culminan con un evento *readEof* tras el cual la única sincronización posible consiste en un evento *close*, con el que finaliza el rol. En este caso

la alternativa entre *read* y *close* tiene carácter global (lo que viene indicado por el operador $\Box$), de manera que el componente conectado a este rol (es decir, quien lee de la tubería) debe de estar preparado tanto para sincronizarse en un evento *read* como en uno *readEof* (dependiendo de si existen o no datos en la tubería).

Además de los roles citados, vemos cómo la especificación del conector *Pipe* de la Figura 1.9 contiene una sección *glue*, también descrita por medio de un protocolo en CSP, que sirve para indicar el comportamiento completo del conector, es decir, para ligar los dos roles, haciendo referencia a eventos tanto de uno como de otro, y estableciendo de esta forma cuál es el entrelazado válido de las operaciones descritas en ambos.

Al igual que sucede en UniCon, la especificación en Wright de un componente compuesto consiste en la enumeración de los componentes y conectores que lo constituyen, y en la descripción de su estructura o arquitectura, es decir, en la indicación de las conexiones entre puertos de sus componentes y roles de sus conectores que lo construyen por medio de la combinación de estos elementos. De esta forma, a partir de la especificación de componentes y conectores simples en Wright, que como hemos visto incluyen una descripción de su comportamiento en CSP, podemos obtener la definición del comportamiento de un elemento compuesto. Para ello, basta con combinar las especificaciones en CSP de los componentes y conectores constituyentes. Esta especificación puede ejecutarse utilizando herramientas de simulación como FDR [Formal Systems, 1992]. Así mismo, es posible analizar los sistemas especificados en Wright. Para ello, basta con comprobar que cada una de sus conexiones es compatible, es decir, que el sistema formado por la combinación de los protocolos descritos en CSP de cada puerto y rol interconectados está libre de bloqueos. Este tipo de comprobaciones puede llevarse a cabo utilizando herramientas como la ya citada FDR.

Para finalizar, debemos señalar que Wright se limita a servir de ayuda a la especificación arquitectónica, ignorando el resto de tareas propias del proceso de desarrollo. Así, no se contemplan en este lenguaje la simulación ni la generación de código a partir de la especificación, ni incorpora mecanismos de reutilización de componentes y conectores, ni de adaptación al cambio. Del mismo modo, tampoco se contempla la existencia de sistemas que presenten una arquitectura dinámica, que pueda evolucionar en tiempo de ejecución.

Darwin

Un lenguaje de descripción de arquitectura que merece especial atención es Darwin [Magee y Kramer, 1996]. A diferencia de UniCon y Wright, que sólo permiten la descripción de estructuras estáticas o inmutables, definidas en tiempo de instanciación o inicialización del sistema, en Darwin es posible describir arquitecturas dinámicas, es decir, que evolucionan durante la ejecución del mismo.

Los dos conceptos básicos que maneja el lenguaje son el de componente y el de *servicio*, no existiendo en Darwin el concepto de conector como una entidad propia del lenguaje. Cada componente indica en su interfaz cuáles son los servicios que proporciona al resto del sistema y también cuáles son los que requiere para su funcionamiento. Cada uno de estos servicios lleva asociado un tipo, a través del que se realizan las comprobaciones de compatibilidad de las conexiones entre componentes.

Darwin proporciona dos mecanismos estrechamente relacionados para la descripción de arquitecturas dinámicas. El primero de ellos es la *instanciación dinámica*, que permite especificar la creación de componentes durante la ejecución del sistema. Estos componentes se organizan en colecciones de tamaño no determinado especificadas dentro de la arquitectura. Cuando se

crea un componente, se añade a la colección y se conecta con los ya existentes siguiendo los patrones de conexión indicados en la especificación arquitectónica.

El segundo de estos mecanismos es la *instanciación perezosa*, por medio de la cual el componente que proporciona un servicio no se instancia hasta que un usuario intenta acceder a dicho servicio. La combinación de la instanciación perezosa con la composición recursiva permite la descripción de estructuras de tamaño variable.

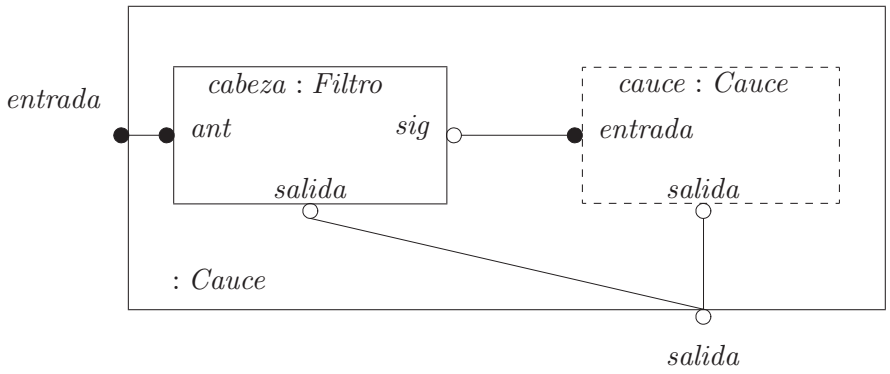


Figura 1.10: Representación gráfica de un cauce en Darwin.

La Figura 1.10 muestra, en su forma gráfica, la codificación en Darwin de un cauce formado por composición recursiva de sucesivos *Filtros*. Cada uno de los filtros presenta una interfaz que proporciona un servicio *ant* (anterior) y precisa de otros dos: *sig* (siguiente) y *salida* (la representación en Darwin de un servicio ofertado se realiza mediante un punto relleno, mientras que los servicios requeridos se indican mediante puntos huecos). Por su parte, la interfaz del cauce indica que este componente proporciona un servicio *entrada*, a la vez que precisa para su funcionamiento de otro, denominado *salida*. Al inicializar el sistema, sólo se instancia un componente *Filtro*, el denominado *cabeza*. El servicio *sig* requerido por el filtro, se conecta al servicio *entrada* proporcionado por el componente *cauce* (obsérvese la composición recursiva de componentes *cauce*). Dicha conexión es perezosa (lo que viene indicado por las líneas discontinuas al representar al componente), con lo que componente *cauce* no se instancia inmediatamente, sino sólo cuando el filtro *cabeza* intenta acceder a su servicio *entrada*. De esta forma, el cauce de componentes *Filtro* se va ampliando durante la ejecución del sistema tanto como sea necesario.

Para finalizar, debemos señalar que Darwin ha sido extendido recientemente para incorporar la posibilidad de analizar propiedades de seguridad y viveza de las especificaciones [Magee et al., 1999], propiedades que se expresan por medio de sistemas de transiciones etiquetadas (LTS) de estado finito. Así mismo el lenguaje contempla la generación de código a partir de las especificaciones. Sin embargo, no se dispone en el lenguaje de mecanismos que faciliten el refinamiento o la reutilización de las especificaciones.

Rapide

Rapide [Luckham et al., 1995] es un lenguaje concurrente basado en eventos para el prototipado de la arquitectura del software. El objetivo fundamental del lenguaje es la obtención de prototipos ejecutables a partir de las especificaciones arquitectónicas. Se pretende que en

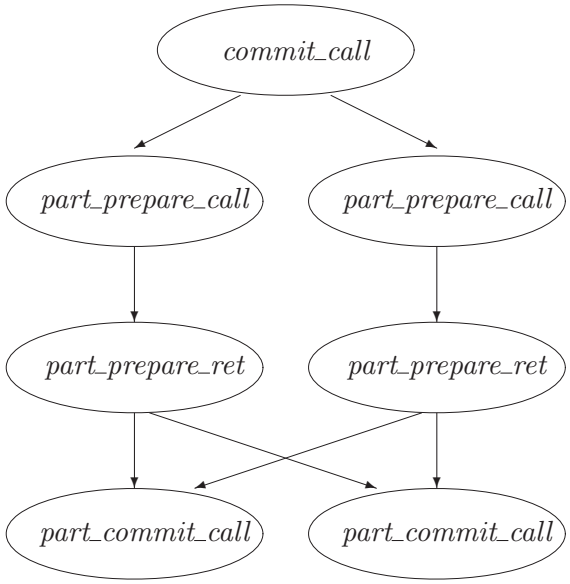


Figura 1.11: *Poset*: Conjunto de eventos parcialmente ordenados.

dichos prototipos aparezcan reflejadas de forma explícita las propiedades de concurrencia, sincronización, flujo de información y temporización de la arquitectura que representan.

Para ello, Rapide adopta un modelo de ejecución basado en los llamados conjuntos de eventos parcialmente ordenados o *posets* (*partially ordered set of events*), que describen las secuencias válidas de eventos que pueden ocurrir en un determinado sistema. Estos eventos se corresponden con la invocación de operaciones entre componentes, y se dividen en *acciones*, que modelan comunicaciones asíncronas, y *funciones*, que modelan comunicaciones síncronas. Mediante este modelo es posible simular las especificaciones escritas en Rapide, además de verificar si las trazas de eventos generados cumplen determinadas restricciones.

La Figura 1.11 muestra uno de estos *posets*, correspondiente a parte del protocolo de compromiso (*commit*) en dos fases utilizado habitualmente para asegurar la atomicidad de las transacciones en sistemas distribuidos. Para este ejemplo supondremos la existencia de un coordinador y dos participantes. Siguiendo este protocolo, a partir del evento inicial *commit_call* —recibido por el coordinador— se generan de forma concurrente dos eventos *part_prepare_call* (dando lugar a dos secuencias independientes de eventos) mediante los cuales el coordinador interroga a los participantes sobre si es posible o no cerrar la transacción. Una vez que ambos participantes responden con un evento *part_prepare_ret*, el coordinador les enviará a ambos el evento *part_commit_call* que compromete la transacción. Las flechas indican dependencias causales entre eventos (así por ejemplo, la figura indica que un evento *part_prepare_ret* sólo puede producirse tras la ocurrencia de un evento *part_prepare_call*), mientras que la ausencia de estas dependencias causales determinan los posibles entrelazados de eventos que pueden ocurrir. De esta forma, un *poset* se corresponde con diversas trazas o secuencias lineales de eventos.

La especificación de componentes en Rapide consta de dos partes: la descripción de la interfaz, que indica tanto las operaciones que el componente ofrece o exporta a su entorno como las que precisa o importa de éste, y su implementación, que puede incluir bien una descripción del protocolo de comportamiento del componente utilizando restricciones y los ya

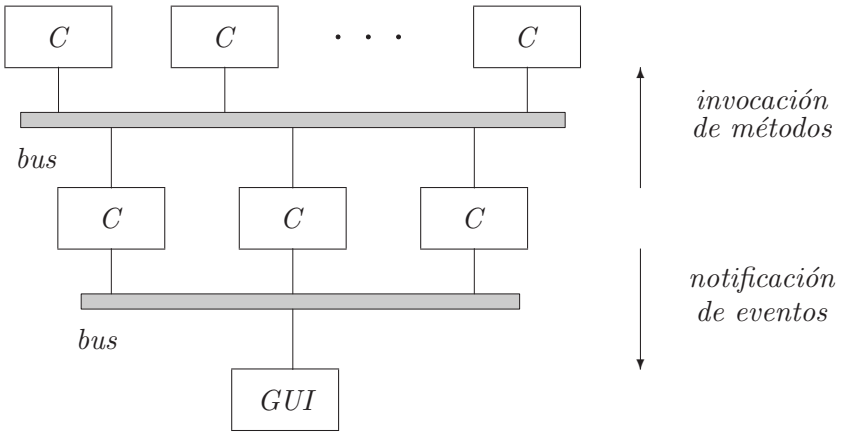


Figura 1.12: Componentes en el estilo C2.

los citados *posets*, o bien un prototipo ejecutable del mismo, o incluso ambas cosas a la vez. El protocolo y las restricciones son utilizados para comprobar que el prototipo ejecutable se ajusta a la especificación del componente o, en caso de que no se proporcione un prototipo, para generarlo automáticamente.

Al igual que ocurre en Darwin, Rapide no incorpora el concepto de conector como parte del lenguaje, sino que modela los sistemas de software como un conjunto de componentes conectados mediante paso de mensajes o eventos. Por tanto, lo que en UniCon o Wright se especificaría como un conector, se modela en Rapide como un componente más —si se considera que tiene la entidad suficiente para ello—, o bien queda incluido en la descripción del protocolo de la interfaz de un componente.

Del mismo modo que en el resto de los lenguajes contemplados, la arquitectura se describe en Rapide por medio de una lista de componentes interconectados entre sí a través de las operaciones importadas y exportadas en las interfaces de dichos componentes. El sistema es capaz de obtener un prototipo ejecutable de la arquitectura descrita.

En Rapide es posible la descripción de arquitecturas dinámicas, si bien de forma restringida puesto que el dinamismo en este lenguaje se limita a la descripción de sistemas que pueden presentar varias configuraciones alternativas [Medvidovic y Rosenblum, 1997]. Por otra parte, el lenguaje incorpora un mecanismo de herencia o subtipado para la reutilización de especificaciones y su adaptación a cambios de los requisitos.

C2

El entorno de desarrollo arquitectónico C2 [Medvidovic et al., 1996] puede ser considerado más como un estilo arquitectónico que como un ADL. Al igual que en el resto de las propuestas, el concepto fundamental subyacente a C2 es que los sistemas de software están formados por componentes que se coordinan y comunican mediante paso de mensajes (en concreto, peticiones de servicios y notificaciones de eventos). Sin embargo, la disposición de los componentes en C2 no es libre, como en el resto de los lenguajes hasta ahora considerados, sino que éstos están organizados en capas, tal como se muestra en la Figura 1.12. Además, C2 está orientado a dar soporte a las necesidades específicas de aplicaciones en las que la interfaz gráfica de usuario (GUI) sea un aspecto fundamental de las mismas.

Aunque el lenguaje dispone de conectores, éstos son de un único tipo, limitándose a la simple transmisión de mensajes entre componentes, para lo que actúan como un *bus* de datos. Los componentes se conectan y coordinan unos con otros por medio de estos buses, que son los que actúan de delimitadores a la hora de establecer los niveles en los que se estructura la arquitectura de una aplicación, existiendo un único conector por cada nivel. Cada componente debe indicar cuáles son las notificaciones de eventos de las que quiere ser informado, así como las invocaciones de métodos que el componente emite hacia los niveles superiores de la arquitectura, y también cuáles son las notificaciones que el componente emite hacia los niveles inferiores y los métodos a los que responde, a partir de invocaciones provenientes así mismo de dichos niveles inferiores. (Cabe señalar que la estructuración en niveles de C2 sigue una ordenación inversa a la habitual, figurando en el nivel inferior el GUI de la aplicación y en los superiores las distintas capas de componentes internos de la misma).

Uno de los aspectos más significativos de este lenguaje es la flexibilidad que presenta a la hora de realizar comprobaciones de tipos en las conexiones entre componentes. Para ello, C2 establece varios niveles de comprobación de tipos, que van desde la *conformidad de interfaz* (comprobación meramente sintáctica de la identidad de la signatura de las interfaces conectadas), y la *conformidad de comportamiento* (que permite que una clase sea un subtipo de otra si preserva la interfaz y el comportamiento de todos los métodos disponibles en el supertipo), hasta llegar a la subclasificación estrictamente monótona, que exige además que el subtipo respete la implementación realizada por el supertipo, sin posibilidad de redefinición (lo que habitualmente se conoce con el nombre de herencia estricta). Mediante una combinación de estos niveles, los autores definen lo que denominan *conformidad de protocolos* [Medvidovic et al., 1998], que va más allá de los métodos individuales, permitiendo especificar restricciones en el orden en el cual los métodos pueden ser invocados. Sin embargo, esta especificación y comprobación del comportamiento de los componentes se lleva a cabo mediante el uso de precondiciones y postcondiciones, lo que si bien en ocasiones resulta sencillo de realizar, se puede acabar convirtiendo en una tarea tediosa que resulte en una especificación complicada y difícil de entender.

Por último, señalar que C2 contempla la generación de código a partir de la especificación, pero no así la verificación de propiedades (al carecer de un soporte formal). Así mismo, existe la posibilidad de describir arquitecturas dinámicas, contemplándose la inserción y eliminación de componentes en la arquitectura, pero está restringido por el estilo de arquitectura en capas que impone el lenguaje.

1.5 Métodos formales usados en Arquitectura del Software

Como acabamos de ver, muchos ADLs tienen una base formal que permite el análisis y verificación de propiedades de los sistemas descritos haciendo uso de estas notaciones. En esta sección haremos un repaso por los principales formalismos utilizados en el campo de la Arquitectura del Software, haciendo especial hincapié en las álgebras de procesos y otros formalismos especialmente adecuados para modelar concurrencia.

1.5.1 Álgebras de procesos

Las álgebras de procesos surgen de la necesidad de dotar de un marco semántico formal a los lenguajes concurrentes, para los cuales las semánticas funcionales existentes hasta la fecha no eran adecuadas. Su origen podemos establecerlo en la década de los ochenta, momento en el que se establecieron las bases de este tipo de formalismos y fueron propuestas las principales representantes de este campo.

Las álgebras de procesos son ampliamente aceptadas para la descripción y análisis de sistemas de software [Baeten, 1990], sobre todo de sistemas concurrentes, como combinación de varios componentes [Moore et al., 1999]. Prueba de ello es su influencia en el desarrollo de diversas técnicas de descripción formal, como por ejemplo LOTOS [ISO, 1989], uno de los estándares de OSI para la especificación de sistemas abiertos, o como el método de especificación y desarrollo RAISE [RAISE, 1992, RAISE, 1995], o incluso en el lenguaje de programación Occam [May, 1985]. La base formal que proporcionan las álgebras de procesos permite el análisis de los sistemas descritos para determinar si satisfacen o no determinadas propiedades de interés.

Desde el punto de vista de las álgebras de procesos, un sistema distribuido es visto como un proceso, probablemente consistente en varios subprocesos, lo que lleva a que la especificación describa un sistema a través de una jerarquía de definiciones de procesos. Un proceso es una entidad capaz de llevar a cabo acciones internas (no observables) y de interactuar con otros procesos que forman su entorno. Las interacciones consisten en acciones elementales de sincronización, que pueden implicar el intercambio de datos, y que se consideran instantáneas, no consumiendo tiempo.

En estos formalismos, la definición de los procesos se realiza por medio de un lenguaje muy simple que refleja las nociones relacionadas con los procesos. La sintaxis utilizada depende del álgebra de procesos concreta que estemos considerando, existiendo diferencias en cuanto a los operadores existentes en las diversas álgebras, pero en general, podemos decir que dicha sintaxis consta de los siguientes elementos:

- Un conjunto de nombres de proceso.
- Un conjunto de nombres de eventos, canales o enlaces, que se utilizan para la sincronización y comunicación entre los procesos.
- Un conjunto de nombres de valores o datos, que son transmitidos a través de los canales.
- Una o varias constantes para representar los procesos inactivos o terminados.
- Acciones internas, que se definen como no observables, en el sentido de que no es posible determinar el momento en que el proceso lleva a cabo dichas acciones.
- Acciones de sincronización de procesos (en un evento) y de comunicación de datos entre dos procesos (por medio del envío y recepción de mensajes a través de un canal o enlace compartido por ambos).
- Operadores de restricción, que permiten delimitar el ámbito de un canal o enlace.
- Operadores de composición secuencial (entre procesos y de acciones con procesos), composición alternativa (no determinista) y composición paralela (tanto para el entrelazado de acciones como para la sincronización).
- Operadores para describir la composición alternativa determinista de procesos, por medio de una condición que determine la elección de una u otra alternativa.

Los términos que se obtienen a partir de la combinación de estos elementos, teniendo en cuenta las reglas de la gramática asociada al álgebra, representan procesos. El significado de estos términos se establece por medio de una semántica operacional estructurada, descrita mediante un sistema de reglas o transiciones, que asocia dichos términos a un comportamiento.

La definición semántica se completa a continuación introduciendo una relación de congruencia entre términos. Dicha relación hace que un proceso (en el nivel semántico) se identifique con una clase congruente de términos del lenguaje. En la mayoría de los casos, esta relación de congruencia se identifica con la igualdad de procesos. Dos procesos se consideran iguales si no son distinguibles por medio de un experimento basado en la observación.

A continuación describiremos brevemente las características de las álgebras de procesos más relevantes.

CCS

El *Calculus of Concurrent Systems* (CCS) fue desarrollado por Milner y colaboradores a partir de trabajos comenzados a finales de los setenta [Milner, 1989]. Se trata de un álgebra de procesos deliberadamente simple, en la que se prescinde de todo lo que no es estrictamente necesario.

En CCS, La sincronización y comunicación entre procesos se modela por medio de la emisión y recepción de mensajes. Por ejemplo, la *acción de salida* $\bar{a}(x)$ representa el envío de un valor x a través del canal a, mientras que $a(y)$ representa la *acción de entrada* complementaria, por la cual se recibe un cierto valor x a través del canal a, con lo que se liga el valor y al valor x recibido.

Por otra parte, la composición secuencial está restringida en CCS a la combinación de acciones y procesos (no de procesos entre sí), convirtiéndose la acción en un *prefijo* del proceso. Así $\alpha.P$ representa al proceso que realiza una acción α y pasa a comportarse como P, donde α puede denotar tanto una acción de entrada o salida como las descritas anteriormente, como una acción interna (representada como τ). En este último caso no se realiza comunicación alguna.

Existe además un operador de composición alternativa no determinista (+), de forma que, siendo α y β acciones de comunicación, y P y Q procesos, $\alpha.P + \beta.Q$ representa el proceso que bien se comporta como $\alpha.P$ o bien se comporta como $\beta.Q$. En este caso, la elección entre una y otra alternativa se realiza de manera *global*, es decir, teniendo en cuenta la disponibilidad del entorno para realizar las acciones complementarias de α o β. Sin embargo, en el proceso $\tau.P + \tau.Q$, la decisión entre una u otra alternativa se realiza de manera *local*, puesto que dicho proceso no tiene en cuenta su entorno a la hora de decidir la realización de una u otra acción interna τ. Tanto en las decisiones globales como en las locales, en caso de que existan varias alternativas posibles, la elección de una u otra se realiza de forma no determinista.

Por el contrario, las alternativas deterministas (condicionales) se expresan mediante la construcción *if-then-else*. Así, siendo P y Q procesos y c un valor, entonces el proceso *if c=1 then P else Q* se comporta como P o como Q dependiendo del valor de c.

El operador (|) indica la composición paralela entre procesos, que pueden sincronizarse mediante el emparejamiento de acciones de comunicación complementarias (es decir, de signo diferente), como por ejemplo en el proceso ($\bar{a}(y).P \mid a(x).Q$). Dicho emparejamiento de acciones de comunicación es observado desde el entorno del proceso como una acción interna τ.

La restricción de ámbito de un canal se indica mediante el operador '\'. Así $P\backslash\{a\}$ indicaría que el canal a se define como privado al proceso P, no siendo conocido, por tanto, por el resto del sistema.

Por último, existe un proceso para especificar el agente inactivo o terminado, el cual se representa como **0**.

Como ejemplo de especificación, consideremos una cola acotada de dos posiciones, en la que el almacenamiento de elementos se hace a través de un enlace *in* y la recuperación a través de un enlace *out*. La especificación de este sistema en CCS es la siguiente:

$$Cell(i,o) = i(x).\overline{o}(x).Cell(i,o)$$
$$Buffer(in,out) = (\ Cell(in,m)\ \mid\ Cell(m,out)\ )\backslash\{m\}$$

Como vemos, hemos especificado dos procesos. El primero de ellos —$Cell(i,o)$— representa una célula de memoria capaz de almacenar un único dato, que recibe a través de su enlace i y retransmite a continuación a través del enlace o. Por otra parte, el proceso $Buffer(in,out)$ representa una cola acotada construida a partir de la composición paralela de dos de estas células. El enlace m que conecta la salida de una célula con la entrada de la siguiente se oculta mediante el operador de restricción, de forma que no pueda ser accedido más que desde el propio proceso *Buffer*.

Con respecto al análisis de las especificaciones escritas en CCS, los conceptos de comportamiento observable y equivalencia de observación juegan un papel fundamental. En primer lugar, se entiende por comportamiento de un proceso el conjunto de las posibles acciones que el mismo puede realizar. Por ejemplo, con respecto a la cola acotada que acabamos de especificar, su comportamiento observable está formado (entre otras) por las trazas o secuencias de acciones:

$$in(a).\tau.\overline{out}(a)\ \ldots$$
$$in(a).\tau.in(b).\tau.\overline{out}(a).\overline{out}(b)\ \ldots$$
$$in(a).\tau.in(b).\tau.\overline{out}(a).in(c).\tau.\overline{out}(b)\ \ldots$$
$$\ldots$$

donde se puede apreciar que las acciones de comunicación entre las dos células del *Buffer* aparecen reflejadas como acciones internas τ.

Desde este punto de vista, la equivalencia (fuerte) entre procesos se define entonces de forma que, aunque su especificación sea diferente, dos procesos se consideren equivalentes si no son distinguibles por ningún experimento basado en la observación, es decir, si ambos presentan exactamente las mismas trazas. Sin embargo, esta noción de equivalencia es muy estricta, puesto que, por lo general, basta con que nos limitemos al *comportamiento observable*, haciendo abstracción de las transiciones internas τ. De aquí se deriva la noción de equivalencia de comportamiento o equivalencia débil [Milner, 1989].

Volviendo al ejemplo de la cola acotada, consideremos ahora una especificación alternativa de la misma:

$$Buffer'(in,out) = in(x).Buffer'_1(in,out,x)$$
$$Buffer'_1(in,out,x) = in(y).Buffer'_2(in,out,x,y)\ +\ \overline{out}(x).Buffer'(in,out)$$
$$Buffer'_2(in,out,x,y) = \overline{out}(x).Buffer'_1(in,out,y)$$

que describimos ahora como un sistema que puede pasar por tres estados distintos, dependiendo de la secuencia de operaciones de entrada y salida que se vayan realizando. Si analizamos ahora las trazas que presenta el proceso $Buffer'$, obtendremos (entre otras) las siguientes:

$in(a).\overline{out}(a)\ \ldots$
$in(a).in(b).\overline{out}(a).\overline{out}(b)\ \ldots$
$in(a).in(b).\overline{out}(a).in(c).\overline{out}(b)\ \ldots$
$\ldots$

Estas trazas sólo difieren de las correspondientes al proceso *Buffer* en la ausencia de transiciones internas τ, por lo que podemos considerar que ambos procesos, *Buffer* y *Buffer'* presentan equivalencia de comportamiento.

Para terminar, diremos que no obstante esta simplicidad sintáctica y conceptual, es posible especificar en CCS comportamientos complejos (si bien de forma no directa), lo que le hace un buen candidato para la descripción y análisis de sistemas concurrentes y distribuidos.

ACP

ACP (*Algebra of Communicating Processes*), desarrollada por Bergstra y colaboradores a inicios de los ochenta [Bergstra y Klop, 1985], es el formalismo del cual el propio término 'álgebra de procesos' se deriva. Este álgebra se basa en una teoría axiomática (un conjunto de ecuaciones) que identifica términos que representan procesos.

Al igual que en CCS, existen en ACP operadores de composición alternativa ($+$), paralela ($||$), y secuencial ($\cdot$), aunque éste último normalmente se omite. La composición secuencial puede utilizarse para componer procesos y no sólo acciones con procesos, como sucede en CCS. Por otro lado, es posible distinguir entre dos formas de terminación, existiendo constantes para especificar la terminación con éxito (*nil*), que también se suele omitir, y el bloqueo (δ).

Este álgebra considera que cada proceso tiene asociado un alfabeto A que representa el conjunto de acciones o eventos que dicho proceso puede realizar. Estas acciones se consideran atómicas y sin duración en el tiempo.

Como ejemplo, podemos modelar en ACP el comportamiento de una *bolsa* binaria, una estructura que contiene valores binarios y en la que, a diferencia de una cola, no importa el orden de inserción de elementos, pero si el número de veces que cada valor binario ha sido insertado en la bolsa. La especificación de este proceso sería la siguiente:

$$Bag = in(0) \cdot (\ out(0)\ \ ||\ \ Bag\ )\ +\ in(1) \cdot (\ out(1)\ \ ||\ \ Bag\ )$$

El proceso *Bag* indica que la bolsa esta capacitada para realizar las acciones $in(0)$ e $in(1)$, que modelan respectivamente la inserción de los dígitos binarios '0' y '1' y que por cada acción de inserción realizada se presenta también disponible la acción de extracción $out(0)$ o $out(1)$ correspondiente. La utilización del operador de composición paralela es la que permite especificar que, a la hora de extraer elementos de la bolsa, el orden de inserción no es significativo.

La sincronización entre procesos se realiza en ACP de acuerdo a una función binaria $\mid$ definida sobre $A \cup \{\delta\}$, de forma que, siendo a y b acciones, $a \mid b$ define el resultado de su sincronización (si las acciones sincronizan), o bien $a \mid b = \delta$ (si no sincronizan).

La comunicación de datos se modela normalmente en ACP por medio de acciones especiales. Así, la acción $r_i(d)$ representa la recepción de un dato d a través de un canal i, mientras que $s_i(d)$ representa la acción complementaria, es decir, el envío de un dato d a través de i. Como es lógico ambas acciones sincronizan, para lo cual se define $r_i(d) \mid s_i(d) = c_i(d)$, donde $c_i(d)$ representa la *comunicación* de d a través de i.

Como ejemplo de especificación en ACP, representaremos una variable v definida sobre un cierto dominio D como un conjunto de procesos V_d, donde $d \in D$, definidos de la siguiente forma:

$$V_d \stackrel{\text{def}}{=} s(v = d) \cdot V_d + \sum_{e \in D} r(v := e) \cdot V_e$$

donde la acción $s(v = d)$ representa el envío de un dato $v = d$, indicando que el valor de la variable v es d, mientras que la acción $r(v := e)$ representa la recepción de un nuevo valor e que se asigna a la variable v.

Podemos ahora especificar dos procesos P_1 y P_2 que realizan operaciones de lectura y escritura sobre la variable compartida v:

$$P_1 \stackrel{\text{def}}{=} s(v := 1) \cdot r(v = 1) \cdot nil$$
$$P_2 \stackrel{\text{def}}{=} s(v := 2) \cdot r(v = 2) \cdot nil$$

donde cada uno de los procesos escribe un valor (1 o 2) en la variable para intentar leer a continuación dicho valor.

Llegados a este punto, podemos determinar fácilmente que el sistema formado por los dos procesos y la variable (establecido un valor inicial para la misma que no es relevante) bloquea, es decir, que:

$$P_1 \mid\mid V_1 \mid\mid P_2 = \delta$$

para lo habríamos de hacer uso de las ecuaciones que definen la semántica de los procesos en ACP. En efecto, resulta evidente que, tras producirse los eventos $c(v := 2) \cdot c(v := 1) \cdot c(v = 1)$ el sistema llegaría al estado:

$$V_1 \mid\mid r(v = 2) \cdot nil$$

en el que no es posible realizar más acciones, puesto que como hemos indicado anteriormente, la función de sincronización $\mid$ está definida como $(\ r(v = 2) \ \mid \ a \ ) = \delta$ para cualquier acción a distinta de $s(v = 2)$, en particular para las acciones $s(v = 1)$, $r(v := 1)$ y $r(v := 2)$ que son las únicas que el proceso V_1 puede realizar.

CSP

CSP (*Communicating Sequential Processes*) [Hoare, 1985] nació como un lenguaje de programación, pero está dotado de un fuerte componente formal que fue desarrollado posteriormente. CSP es sin duda la más compleja de las álgebras de procesos consideradas aquí.

En primer lugar, en CSP se distinguen tres formas de terminación: los procesos $STOP$, $SKIP$ y RUN. El proceso $STOP$ modela las situaciones de interbloqueo, es decir, representa a un proceso que no es capaz de realizar ninguna acción, mientras que $SKIP$ modela la terminación con éxito, para lo cual se considera que realiza una acción especial '$\surd$' y acaba. Por su parte, RUN representa a un proceso divergente, es decir, que es capaz de realizar indefinidamente cualquier acción y que, por lo tanto, no acaba.

Así mismo, se puede especificar tanto la composición secuencial de una acción y un proceso, como en $a \rightarrow P$, como la composición secuencial de procesos, como en $P; Q$.

Además, se utilizan acciones distintas para representar la sincronización y la comunicación entre procesos. En CSP la sincronización de dos procesos tiene lugar en un evento sin signo. Por ejemplo, siendo P y Q procesos y a un nombre, los procesos $a \rightarrow P$ y $a \rightarrow Q$ pueden sincronizarse en el evento a, del que, como vemos, no se indica signo. Por el contrario, la comunicación entre procesos implica intercambio de datos, por lo que se debe indicar que uno de los procesos realiza una acción de salida y el otro la acción de entrada complementaria. De este modo, si P y Q son procesos, a es un canal y x e y son valores, los procesos $a!x \rightarrow P$ y $a?y \rightarrow Q$ pueden comunicarse a través de dicho canal, enviando el primero de ellos el valor x al segundo, que instancia su valor y al x recibido.

Existen así mismo varios operadores para la composición alternativa de procesos: la elección (representada mediante $\mid$), la decisión global (representada mediante $\Box$) y la decisión local (representada mediante $\sqcap$). Por su parte, al componer procesos en paralelo, esta composición puede definirse como un entrelazado de las acciones ambos procesos (lo que se indica mediante el operador $|||$), o como una sincronización (mediante el operador $||$), en cuyo caso se impone la sincronización de los eventos comunes de ambos procesos.

Como ejemplo de especificación en CSP, consideremos un pila no acotada, que denominaremos *Stack*, en la que podemos almacenar elementos a través de un canal de entrada *push* y recuperarlos a través del canal de salida *pop*. Supongamos que el proceso termina una vez que la pila queda vacía. De acuerdo con esto, su especificación sería la siguiente:

$$
\begin{aligned}
Stack &= Stack_{<>} \\
Stack_{<>} &= push?x \rightarrow Stack_{<x>} \\
Stack_{<x>} &= pop!x \rightarrow SKIP \ \Box \ push?y \rightarrow Stack_{<y>};\ Stack_{<x>}
\end{aligned}
$$

Como podemos observar, la pila está inicialmente vacía, siendo representada por el proceso $Stack_{<>}$, y la única operación que podemos realizar sobre ella es la de almacenar un elemento x, que la pila recibe a través del canal *push*. A continuación, la pila cambia de estado, siendo ahora especificada mediante el proceso $Stack_{<x>}$. Éste presenta dos comportamientos alternativos, combinados mediante el operador $\Box$, lo que indica que es una decisión global, es decir, que es el entorno de la pila quien determina cuál de las dos operaciones se va a realizar. En caso de que se produzca una nueva escritura —$push?y$— se crea un nuevo proceso $Stack_{<y>}$ que se compone secuencialmente con el que ya existe. En cambio, si lo que se realiza es una lectura —$pop!x$— el proceso acaba con éxito ($SKIP$).

Una vez especificado el sistema, podemos proceder a su análisis. Pero antes, debemos presentar primero dos nociones fundamentales en CSP. En primer lugar, tenemos el concepto de *alfabeto* de un proceso P, que se define como el conjunto de acciones en las que éste es capaz de sincronizarse, y se representa como αP. Es posible distinguir entre el alfabeto de entrada y de salida —$\mathbf{i}P$ y $\mathbf{o}P$, respectivamente— lo que indica además el signo de estas acciones. Por ejemplo, para la pila que acabamos de especificar, tenemos que:

$$
\begin{aligned}
&\mathbf{i}Stack = \mathbf{i}Stack_{<>} = \{\ push\ \} \\
&\mathbf{o}Stack = \mathbf{o}Stack_{<>} = \varnothing \\
&\mathbf{i}Stack_{<x>} = \{\ push\ \} \\
&\mathbf{o}Stack_{<x>} = \{\ pop\ \}
\end{aligned}
$$

Una vez especificado su alfabeto, podemos determinar cuáles son las trazas de un proceso P (lo que se representa como $\mathbf{t}P$), donde cada traza es una secuencia de acciones en las que el proceso puede sincronizarse. De nuevo en el caso de la pila, tenemos que:

$$\mathbf{t}Stack = \mathbf{t}Stack_{<>} = \mathbf{t}(push?x \rightarrow Stack_{<x>}) = \{ < push.x > \frown t \mid t \in \mathbf{t}Stack_{<x>} \}$$

donde '$\frown$' representa la concatenación de trazas. Por su parte,

$$\begin{aligned}\mathbf{t}Stack_{<x>} &= \mathbf{t}(\ pop!x \rightarrow SKIP \ \Box\ push?y \rightarrow Stack_{<y>};\ Stack_{<x>}\ ) \\ &= \{ < pop.x > \frown \mathbf{t}SKIP, < push.y > \frown \mathbf{t}(Stack_{<y>};\ Stack_y) \} \\ &= \{ < pop.x, \surd >, < push.y > \frown \mathbf{t}Stack_{<y>};\ Stack_y \}\end{aligned}$$

lo que nos llevaría a ir construyendo el siguiente conjunto de trazas:

$$\{ < push.x, pop.x, \surd >, < push.x, push.y, pop.y, pop.x, \surd > \ldots \}$$

que sirve de base para realizar cualquier análisis del sistema (por ejemplo, determinar la equivalencia o la inclusión entre procesos, situaciones de bloqueo, etc.) y también para llevar a cabo una simulación de la especificación.

El cálculo π

Dentro de las álgebras de procesos, el cálculo π [Milner et al., 1992] ocupa un lugar destacado, al estar especialmente indicado para la descripción y análisis de sistemas concurrentes cuya topología sea dinámica. Estos sistemas se representan en el cálculo π mediante colecciones de procesos o *agentes* que interactúan por medio de enlaces o *nombres*. Estos nombres pueden considerarse como canales bidireccionales compartidos por dos o más agentes, y actúan como *sujetos* de la comunicación, en el sentido de que la comunicación se realiza a través de ellos.

Para una descripción detallada de los aspectos técnicos del cálculo, nos remitimos a la Sección 2.1. Aquí nos limitaremos a señalar que el cálculo π permite representar de forma directa la movilidad [Engberg y Nielsen, 1986], lo que se consigue mediante el envío de nombres de canales como argumentos u *objetos* de los mensajes. De hecho, el cálculo no distingue entre canales y datos; todos ellos son considerados genéricamente *nombres*. Cuando un agente recibe un nombre, puede usar este nombre como sujeto para una transmisión futura, lo que proporciona una forma sencilla y efectiva de reconfigurar el sistema.

Así por ejemplo, $a(x).\bar{x}(z)$ representa a un proceso que recibe un nombre de enlace a través del enlace a, substituyendo su nombre x por el del enlace recibido, y utiliza a continuación dicho enlace para enviar por él un nombre z. Este doble uso de los nombres de enlace, como sujeto y objeto en las acciones de comunicación no es posible en el resto de formalismos considerados, que distinguen rigurosamente entre nombres de canales y nombres de valores. Es precisamente este tratamiento homogéneo de los nombres en el cálculo π lo que permite construir un álgebra muy simple, pero también muy potente en la que se pueden especificar de forma elegante sistemas que presenten una topología dinámica.

Como ejemplo de especificación en el cálculo π, veamos cómo se definen en este álgebra valores y estructuras de datos. Empezaremos por especificar los valores lógicos mediante dos procesos:

$$\begin{aligned}False(x) &= \overline{x}\textsc{False}\ .\mathbf{0} \\ True(x) &= \overline{x}\textsc{True}\ .\mathbf{0}\end{aligned}$$

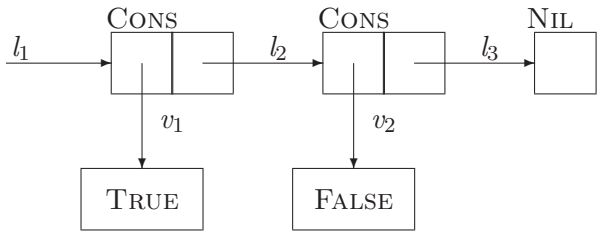


Figura 1.13: Esquema representando una lista de dos valores lógicos.

El acceso a los valores lógicos se hace a través de un enlace, al que podemos considerar como una *referencia* al valor en cuestión. A través de dicho enlace, el proceso lector recibirá (en forma de constante True o False) el valor de que se trate. Obsérvese que una vez emitido el valor, los procesos acaban, pero que para dotarlos de persistencia bastaría con hacerlos recursivos, admitiendo entonces un numero indefinido de consultas.

Podemos ahora especificar listas de tamaño arbitrario, a partir de los constructores habituales Cons y Nil:

$$Nil(x) = \overline{x}\ \text{Nil}.\ \mathbf{0}$$
$$Cons(x,h,t) = \overline{x}\text{Cons}.\ \overline{x}h.\ \overline{x}\ t.\ \mathbf{0}$$

donde $Nil(x)$ es un proceso que representa a la lista vacía, mientras que $Cons(x,h,t)$ representa a la lista en cuya cabeza tenemos almacenado el valor h y cuyo resto es la lista t. En ambos procesos, el acceso al valor almacenado se hace a través de la referencia x.

La Figura 1.13 muestra la estructura que representa a una lista que contiene dos elementos lógicos: True y False. Esta lista estaría especificada en el cálculo π por los procesos:

$$\begin{aligned}(l_2,l_2,l_3,v_1,v_2)(\ & Cons(l_1,v_1,l_2)\ \ |\\ & Cons(l_2,v_2,l_3)\ \ |\\ & Nil(l_3)\ \ |\\ & True(v_1)\ \ |\\ & False(v_2)\ )\end{aligned}$$

donde l_1 sería la referencia a la lista, a través de la cual se puede realizar su recorrido elemento a elemento.

Obsérvese cómo ya en un ejemplo tan sencillo como el descrito hemos hecho uso de la movilidad característica del cálculo π. En efecto, a través del enlace l_1 que representa la lista, recibiremos otros enlaces que hacen referencia a su vez a otros valores, concretamente el enlace v_1 a través del cual accederemos al valor True almacenado en la primera posición de la lista, y el enlace l_2 que referencia a su vez a la lista situada en la cola de l_1.

En la codificación anterior hemos hecho uso de constantes globales para representar tanto los valores lógicos como los constructores de la lista. Sin embargo, no son necesarias dichas constantes en el cálculo π, como muestra la codificación alternativa de los valores lógicos que ofrecemos a continuación:

$True(x) = (u, v)\ \overline{x}u.\ \overline{x}v.\ \overline{x}u.\ \mathbf{0}$
$False(x) = (u, v)\ \overline{x}u.\ \overline{x}v.\ \overline{x}v.\ \mathbf{0}$

y en la que, sin hacer uso de constante global alguna, el proceso que explore el valor lógico podrá determinar si se trata del valor verdadero o falso, en función de si el tercer enlace recibido es igual al primero o al segundo.

Prueba de la adecuación del cálculo π para la especificación de sistemas de software, especialmente en el caso de sistemas distribuidos y dinámicos, es su utilización en distintos propuestas, bien sea como base formal para la definición de otras notaciones, o bien para describir la semántica de diversos modelos y lenguajes. Así mismo, ha dado lugar al desarrollo de diversas herramientas de análisis y simulación.

Uno de los primeros trabajos en este terreno consistió en la codificación del cálculo λ por medio del cálculo π. Este trabajo resulta especialmente interesante por el hecho de ser el cálculo λ un modelo teórico ampliamente aceptado para representar la computación secuencial, mientras que el cálculo π pretende alcanzar el mismo *status* en el campo de la computación concurrente y distribuida. En [Milner, 1992, Sangiorgi, 1992] se muestra cómo esta codificación puede realizarse de una forma relativamente sencilla, incluso con un subconjunto del cálculo π. En estos trabajos se hacen evidentes los paralelismos entre ambos formalismos, demostrándose de este modo que la expresividad del cálculo π no se limita a la descripción de los aspectos de coordinación y comunicación entre procesos, sino que es capaz de representar también computaciones sobre valores de datos y procesamiento funcional de los mismos.

En la misma línea de establecer relaciones entre el cálculo π y otros formalismos, debemos citar la implementación del cálculo π por medio de un intérprete de reescritura [Viry, 1996b]. El enfoque de este trabajo se basa en considerar el sistema de transiciones del cálculo π, convenientemente adaptado, como un sistema de reglas de reescritura. El objetivo final de estos trabajos es dotar al intérprete de reescritura ELAN de capacidades de entrada/salida, lo que se lleva a cabo incorporando a dicho intérprete primitivas para la lectura y escritura de datos a través de canales externos que siguen el modelo de comunicación del cálculo π [Viry, 1996a], lo que resulta en una combinación de ambos formalismos.

Si consideramos ahora los lenguajes derivados directamente del cálculo π, debemos hablar en primer lugar de Pict [Pierce y Turner, 1999], un lenguaje experimental cuya semántica se define por medio de su traducción a una versión simplificada y asíncrona del cálculo. Por este motivo, podemos considerar a Pict como un intento de convertir el cálculo π en un verdadero lenguaje de programación. Aparte de los operadores básicos para el envío y recepción de mensajes presentes en cualquier notación derivada de un álgebra de procesos, Pict incorpora toda una colección de tipos básicos y compuestos, además de las operaciones aritméticas, lógicas y de entrada/salida habituales en los lenguajes de programación. Existe una implementación de este lenguaje que se ejecuta en un entorno monoprocesador [Pierce, 1997].

Una aproximación similar es la seguida en el desarrollo de Nepi [Horita y Mano, 1996], que está basado en el cálculo $\nu\pi$, una variante del cálculo π que se demuestra equivalente a este último, pero que ofrece ventajas a la hora de realizar una implementación distribuida. Aunque la sintaxis de Nepi no se aleja mucho de la utilizada habitualmente en el contexto de las álgebras de procesos (lo que lo aleja de ser considerado un lenguaje de programación), su característica más relevante es precisamente que se trata de un lenguaje distribuido, susceptible por tanto de ser utilizado en un entorno multiprocesador.

También debemos considerar dentro este grupo a Piccola [Lumpe, 1999], un lenguaje para la construcción de sistemas concurrentes y distribuidos, especificados por medio de guiones

(*scripts*) composicionales. Piccola está basado en el cálculo $\pi\mathcal{L}$, una variante asíncrona del cálculo π de la que se ha implementado un intérprete sobre la Máquina Virtual Java (JVM), lo que permite que las aplicaciones desarrolladas sobre este entorno se puedan ejecutar en diversas plataformas hardware y software. Los guiones describen cómo combinar los componentes almacenados en un repositorio que contiene tanto una definición de la interfaz del componente como una implementación del mismo. La implementación consiste en un fichero objeto que contiene el código binario que debe ser combinado por el ensamblador de componentes para obtener un sistema ejecutable. Las definiciones de interfaz son utilizadas por el compilador de Piccola para realizar análisis estáticos cuando los componentes correspondientes son utilizados dentro de un guión.

Aparte de los trabajos citados, el cálculo π ha sido utilizado también para definir la semántica formal de lenguajes no directamente relacionados con este formalismo. Uno de los primeros de estos trabajos es la elaboración de la semántica del lenguaje POOL, un sencillo lenguaje orientado a objetos que utiliza un mecanismo de cita (*rendez vous*) para la interacción entre objetos. La semántica de POOL se presenta en [Walker, 1991] mediante la definición de esquemas de traducción de las estructuras sintácticas del lenguaje a procesos en el cálculo π. Estos esquemas definen cómo los objetos, variables, expresiones, métodos y clases se representan mediante enlaces en el cálculo π, enlaces que sirven de referencia para acceder a dichos elementos. Para que un objeto invoque un método de otro, en primer lugar debe conocer su referencia, consistiendo entonces la invocación en el envío de un mensaje a través de dicha referencia. El objeto receptor pasa a realizar la operación solicitada, mientras que el objeto emisor suspende su ejecución hasta obtener una respuesta (un valor de retorno) de la operación invocada.

Un enfoque muy similar al que acabamos de citar es el seguido para definir la semántica de $\pi o\beta\lambda$ [Jones, 1993], un lenguaje orientado a objetos que admite concurrencia entre objetos aunque no entre los métodos de un objeto.

Incluso dentro del propio campo de los ADLs se ha utilizado el cálculo π. En efecto, en [Radestock y Eisenbach, 1994, Magee et al., 1995] se describe por medio de este álgebra de procesos la semántica de los distintos tipos de conexiones entre componentes que ofrece el lenguaje Darwin, al que ya hemos hecho referencia en la Sección 1.4.5. Para ello, cada uno de los servicios ofrecidos o requeridos por un componente se define por medio de un proceso en el cálculo π, y se asocia a dicho proceso un enlace o canal que sirve para hacer referencia al servicio. Por último, las conexiones entre los servicios de los diferentes componentes se modelan de manera que establecen una comunicación entre los mismos en la forma indicada por la arquitectura del sistema.

Además de lo anterior, el cálculo π ha sido utilizado extensamente para representar formalmente diversos modelos de objetos y componentes. Así, en [Walker, 1995] se presenta una codificación en el cálculo de un modelo computacional de objetos que es el punto de partida de los ya citados trabajos [Walker, 1991] y [Jones, 1993], mientras que en [Henderson, 1997] se utiliza el cálculo π para describir un modelo de componentes distribuidos. Del mismo modo, las plataformas CORBA y COM han sido modeladas en el cálculo π por [Gaspari y Zabattaro, 1999] y [Feijs, 1999], respectivamente. En ambos trabajos se saca partido de las características de movilidad del cálculo para modelar la gestión de interfaces en estas plataformas, resolviendo de forma elegante tanto las conexiones dinámicas entre componentes como las suscripciones a eventos. Así mismo, el cálculo π ha sido propuesto para la extensión del lenguaje de descripción de interfaces de CORBA [Canal et al., 2000].

Por último, existen varias herramientas de análisis y simulación para el cálculo π. Entre ellas cabe citar *Mobility Workbench* (MWB) [Victor, 1994] y *e-pi* [Henderson, 1998]. Estas herramientas son capaces de simular la ejecución de especificaciones de procesos en el cálculo π, produciendo como resultado las trazas o secuencias de eventos generadas por el sistema especificado. También son capaces de realizar diversos análisis sobre las especificaciones, por ejemplo, de verificar la equivalencia entre procesos.

Limitaciones de las álgebras de procesos

Como ya hemos indicado, las álgebras de procesos son formalismos comúnmente aceptados para la especificación de sistemas de software complejos. Incluso aunque dichos sistemas no tengan una naturaleza inherentemente concurrente, resulta natural considerarlos como un conjunto de partes que interactúan entre sí [Milner, 1989]. No obstante esta adecuación, entre los puntos débiles de las álgebras de procesos podemos citar los siguientes [Bruns, 1997]:

En primer lugar, debemos señalar la imposibilidad de expresar propiedades de tiempo real. En efecto, en los formalismos considerados, la única forma de representar un lapso de tiempo es mediante una acción interna τ (y eso en aquellas álgebras que contemplen dichas acciones). De esta forma, en CCS el proceso $\tau.P$ realiza una acción interna τ y pasa a comportarse 'a continuación' como el proceso P, lo que podría interpretarse como el transcurso de un instante (discreto) de tiempo. Pero, por definición este tipo de acciones internas no son observables, en cuanto a que no podemos determinar su duración ni cuando se van a producir, por lo que no son de hecho una forma adecuada de representar el tiempo.

No obstante, existen diversas variantes de estas álgebras que incorporan construcciones para manejar el tiempo real. Entre ellas, podemos citar *Timed CSP* [Reed y Roscoe, 1988, Davies y Schneider, 1992], una extensión de CSP que incorpora guardas temporales, temporizadores y transferencias de control ligadas al tiempo. También han sido propuestas extensiones de tiempo real para CCS y ACP en [Moller y Tofts, 1990] y [Baeten y Bergstra, 1991], respectivamente.

En segundo lugar, está el hecho de que el modelo de comunicación que incorporan normalmente las álgebras de procesos es síncrono, es decir, la comunicación consiste en emparejar acciones de entrada y salida de dos procesos que están compuestos en paralelo. Aunque satisfactorio si nos movemos en un nivel de abstracción alto, este modelo no se adapta bien a las características de los sistemas reales en los que existe una red de comunicación en donde se producen retrasos e incluso pérdida de mensajes, y en los que, por tanto, las comunicaciones son asíncronas. Existen diversas álgebras de procesos que permiten modelar comunicaciones asíncronas, entre las que podemos citar RPT (*Receptive Process Theory)* [Josephs, 1992], una variante de CSP en la que se parte del supuesto de que todos los procesos son *receptivos* (es decir, que aceptan cualquier mensaje en cualquier momento, aunque eso les lleve a un estado de error). Un enfoque distinto es el seguido en la *Theory of Asynchronous Processes* (TAP) [Josephs et al., 1989], en la que cada proceso dispone de una cola infinita conde almacena los eventos y mensajes de entrada para procesarlos posteriormente.

Dos versiones asíncronas del cálculo π han sido presentadas en [Honda y Tokoro, 1991] y [Boudol, 1992]. La primera de ellas está basada en el uso de colas de mensajes entrantes, al igual que ocurre en TAP. En cambio, en la propuesta de Boudol, los mensajes se consideran como agentes o procesos elementales y se elimina el carácter bloqueante que presentan las acciones de salida en el cálculo síncrono, con lo que el proceso que realiza la acción de salida no necesita esperar a que exista otro dispuesto a realizar la acción complementaria. En ambas propuestas,

el cálculo π asíncrono carece de composición alternativa (+). Se ha demostrado que es posible codificar el cálculo síncrono (prescindiendo del operador de composición alternativa) en la versión asíncrona, e incluso que la codificación de este último operador es posible cuando se utilizan únicamente alternativas guardadas por acciones de entrada [Nestmann y Pierce, 1996], pero no cuando aparecen alternativas con guardas de salida o mixtas [Palamidessi, 1997], por lo que habremos de considerar la expresividad del cálculo π asíncrono inferior a la del síncrono.

En tercer lugar, está la dificultad de modelar componentes que se sincronicen por medio de un reloj global. Excepto por el carácter síncrono de las comunicaciones entre procesos, cada uno de ellos progresa a su propio ritmo, con independencia del resto, y tal como hemos visto sólo es posible sincronizar simultáneamente, por medio de la comunicación, dos procesos, pero no más. También por este motivo, modelar la difusión de un mensaje a un grupo de procesos presenta cierta dificultad, al menos si pretendemos que esta difusión tenga carácter síncrono. No obstante, existen variantes de estas álgebras, como SCCS (*Synchronous CCS*), en el que en cada transición se producen varias acciones elementales, una por cada proceso del sistema, y que son más adecuadas para representar este tipo de sistemas síncronos.

Por último, otro inconveniente que suelen presentar la mayoría de las álgebras de procesos es la imposibilidad de modelar sistemas que tengan una estructura dinámica, en los que las interconexiones entre componentes se puedan crear y destruir durante la ejecución del sistema. Aunque esta deficiencia es común a álgebras de *primera generación*, como CCS, CSP y ACP, formalismos posteriores, como el cálculo π y otros surgidos a partir de él, permiten representar de forma sencilla sistemas que tienen una configuración cambiante. Recientemente ha sido propuesto el cálculo de ambientes (*Ambient Calculus*) [Cardelli y Gordon, 2000] con el objetivo de describir tanto la movilidad de procesos o agentes como de dispositivos. El cálculo de ambientes está basado en la noción de *ambiente* como espacio delimitado, de carácter jerárquico y móvil, donde tienen lugar las computaciones, y en su formulación se han integrado conceptos provenientes fundamentalmente del cálculo π asíncrono y la *máquina química abstracta* (formalismo este último que describimos brevemente en la Sección 1.5.2).

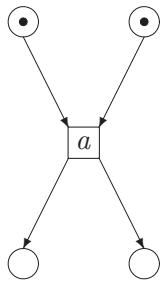
1.5.2 Otras aproximaciones formales

Por sus características, en particular su expresividad a la hora de describir sistemas concurrentes, las álgebras de procesos son el formalismo frecuentemente utilizado en el campo de la Arquitectura del Software. No obstante, tal y como veremos en la Sección 2.5, diversos trabajos proponen la utilización de otros soportes formales para la especificación y análisis arquitectónicos. Está sección está dedicada a describir los más significativos de entre ellos.

Redes de Petri

Sin duda, el más antiguo de los formalismos utilizados para la especificación y análisis de sistemas concurrentes y distribuidos son las redes de Petri, una extensión de la teoría de autómatas secuenciales para admitir concurrencia que tiene su origen en los años sesenta.

Una red de Petri se define formalmente como una tupla $(A, Pl, \longrightarrow, M_0)$ donde A es el alfabeto de comunicación de la red, Pl es un conjunto de nodos o *lugares*, $\longrightarrow$, denominada *función de transición*, es una función parcial de $\mathcal{P}(Pl) \times (A \mid \cup\{\tau\})$ en $\mathcal{P}(Pl)$ y M_0 es el lugar o nodo inicial.



$a \rightarrow SKIP \;||\; a \rightarrow SKIP$

Figura 1.14: Red que representa la sincronización dos procesos y su equivalente en CSP.

Como se puede observar, la diferencia respecto a un autómata secuencial está en la definición de la función de transición, definida aquí sobre subconjuntos de lugares, de forma que podemos representar la sincronización entre dos (o más) procesos concurrentes, como sucede por ejemplo en la red de la Figura 1.14, en la que los estados están representados mediante círculos (◯), los estados iniciales (concurrentes) están marcados con un punto (⊙) y las transiciones aparecen como multiarcos etiquetados con la acción del alfabeto que las dispara, representada en un cuadrado (□). Esta red equivale al proceso CSP representado a la derecha de la figura, en la que dos procesos concurrentes se sincronizan en un evento a.

Respecto a las posibilidades de análisis, es posible determinar la equivalencia de redes, la bisimulación (en el sentido en que se utiliza este término en las álgebras de proceso), realizar análisis de alcanzabilidad de lugares, simular una red, etc.

Entre las ventajas de las redes de Petri podemos apuntar el que las transiciones del sistema obedecen a un modelo de concurrencia muy apropiado para describir, por ejemplo, aplicaciones de tiempo real. Por el contrario, entre sus puntos débiles debemos citar en primer lugar la falta de composicionalidad, debido a que modelan todo el sistema como un único bloque, en vez de representar sus componentes de forma independiente (tal como sucede, por ejemplo, con las álgebras de procesos). Tampoco disponen de mecanismos de abstracción que permitan tratar los procesos como 'cajas negras', ocultando su estructura y comunicaciones internas [Olderog, 1991].

La máquina química abstracta

La *máquina química abstracta* (CHAM, de sus iniciales en inglés) [Berry y Boudol, 1992] es un modelo que ofrece una semántica puramente concurrente, siguiendo un símil químico. A grandes rasgos, podemos decir que la CHAM representa un sistema como una *solución*, en la cual flotan una serie de *moléculas* que interactúan unas con otras de acuerdo a unas *reglas de reacción*. Estas reacciones se pueden producir en paralelo, siempre y cuando involucren a conjuntos disjuntos de moléculas. Existe además un mecanismo básico de abstracción, consistente en definir *membranas* que permiten aislar las soluciones unas de otras.

Más formalmente, una CHAM consiste en una tupla $(M, G, \longrightarrow)$, donde M es un conjunto de moléculas, G es una gramática que determina la sintaxis según la cual se construyen las moléculas, y $\longrightarrow$ es una relación de transición de estados, definida sobre $\mathcal{P}(M)$, es decir sobre soluciones.

El modelo establece una serie de reglas o leyes generales, que definen la semántica de las soluciones, las reacciones y las membranas, y una serie de reglas específicas que rigen el funcionamiento de cada máquina concreta. Esta es quizá la característica más relevante de este formalismo: el que no esté ligado a un sistema de transiciones o reglas establecido de antemano, sino que las reglas de reacción son específicas de cada máquina. De esta forma se logra una gran flexibilidad, lo que permite especificar mediante la CHAM otros formalismos.

Por ejemplo, para definir como reacción química de una máquina la sincronización de procesos por medio de un evento (tal como existe en CCS) habría que especificar la siguiente regla:

$$a.P,\ \bar{a}.Q \longrightarrow P,\ Q$$

que indica que en cualquier solución en la que estén presentes las moléculas $a.P$ y $\bar{a}.Q$ reacciona, produciendo como resultado una solución con las moléculas P y Q.

De este modo, definiendo los operadores de un álgebra de procesos y su sintaxis como parte de la gramática de una máquina química y su sistema de transiciones como reglas o reacciones de dicha máquina, se pueden especificar las distintas álgebras de procesos [Boudol, 1994], incluyendo un subconjunto del cálculo π, en su variante asíncrona, por lo que podemos decir que la CHAM comparte la posibilidad de expresión de movilidad propia de este formalismo (si bien en la CHAM dicha expresión se lleva a cabo de forma indirecta).

Lenguajes de coordinación

Los lenguajes de coordinación se basan en un modelo de programación que separa los aspectos de *computación* y *coordinación* de los sistemas concurrentes. De esta forma, mientras que el modelo de computación permite la construcción de actividades computacionales que se desarrollan paso a paso siguiendo un único flujo de control, el modelo de coordinación permite la combinación de actividades computacionales separadas [Gelernter y Carriero, 1992]. Para lograr este objetivo, un lenguaje de coordinación debe proporcionar operaciones para crear actividades computacionales y también para permitir la comunicación entre ellas. Esta comunicación se realiza a través de un espacio de datos compartido por los distintos procesos que forman el sistema. Un lenguaje de coordinación de este tipo se combina con un lenguaje de computación para conseguir de esta forma expresar computaciones concurrentes.

El más extendido de los lenguajes de coordinación es Linda [Gelernter, 1985], mediante el cual un sistema de software se implementa como un conjunto de procesos que se coordinan y comunican escribiendo y leyendo unos datos o *tuplas* en un espacio común. En Linda se distinguen dos tipos de tuplas: las pasivas, que sólo contienen datos, y las activas, que representan los procesos a ejecutar. Para acceder a un espacio de tuplas, los procesos disponen de varias primitivas:

out(***tupla***(***datos***)) coloca una tupla pasiva *tupla*(*datos*) en el espacio de tuplas.

in(***patrón***) extrae una tupla pasiva *tupla*(*datos*) del espacio de tuplas, siendo *patrón* una tupla pasiva que contiene un patrón de búsqueda que ha de satisfacer la tupla extraída.

rd(***patrón***) es igual a *in*(*patrón*), salvo que extrae sólo una copia de la tupla pasiva *tupla*(*datos*), que permanece en el espacio de tuplas.

***eval*(*proceso*)** coloca una tupla activa (un proceso) en el espacio de tuplas. Dicho proceso comienza a ejecutarse concurrentemente con el proceso que la coloca.

Las primitivas *eval* y *out* son no bloqueantes, mientras que *in* y *rd* sí bloquean al proceso que las invoca, en caso de no haber concordancia entre ninguna tupla del espacio con el patrón pedido, y hasta que algún proceso coloque en el espacio una tupla que satisfaga ese patrón.

Aunque en su origen los lenguajes de coordinación del tipo de Linda carecían de una base formal, posteriormente se ha comenzado a explorar su semántica, utilizando para ello diversos modelos formales [Ciancarini et al., 1995, Brogi y Jacquet, 1999]. En estos trabajos se destaca la relación entre estos lenguajes de coordinación y los lenguajes lógicos concurrentes como Parlog [Clark y Gregory, 1986] o *Shared Prolog* [Brogi y Ciancarini, 1991]. Todos estos lenguajes formarían parte de los enfoques imperativo y lógico, respectivamente, de la programación concurrente con restricciones (*Concurrent Constraint Programming, CCP*), desarrollada por Saraswat y colaboradores [Saraswat y Rinard, 1991].

Por tanto, aunque podemos considerar a CCP como un derivado de la programación lógica, al intentar extender ésta para que admita concurrencia, este formalismo se encuentra también próximo al punto de vista de las álgebras de procesos [de Boer y Palamidessi, 1992]. En efecto, CCP postula que un sistema de software se representa mediante un agente, de forma similar a como se haría utilizando un álgebra de procesos, y de acuerdo con la siguiente sintaxis:

Guardas $g ::= \mathbf{tell}(c) \mid \mathbf{ask}(c)$

Agentes $A ::= \sum_i g_i \rightarrow A_i \mid A \mid\mid A \quad \exists_x A \mid \mathbf{Stop} \mid p(\tilde{x})$

La semántica operacional de CCP viene dada por un sistema de transiciones similar al de las álgebras de proceso, aunque de manera informal podemos decir que las acciones básicas se representan mediante las guardas $\mathbf{ask}(c)$ y $\mathbf{tell}(c)$, donde c es una *restricción finita*, un elemento de un cierto conjunto $\mathcal{C}$. Estas acciones operan sobre un *almacén* común de elementos de $\mathcal{C}$. La acción $\mathbf{ask}(c)$ comprueba la presencia de c en el almacén, sin modificarlo. Diremos que la acción $\mathbf{ask}(c)$ está *disponible*, si c está en el almacén (si seguimos un enfoque imperativo) o bien se deriva del mismo (si seguimos en cambio un enfoque lógico), mientras que *falla* si contradice el contenido del almacén, *suspendiendo* en el resto de los casos. Por su parte, la acción $\mathbf{tell}(c)$ está disponible si c no está (o no contradice) el contenido del almacén, y su ejecución añade c al mismo, fallando en caso contrario.

La alternativa con guardas $\sum_i g_i \rightarrow A_i$ selecciona de manera no determinista una acción g_i que esté disponible, para pasar a continuación a comportarse como A_i. Si ninguna guarda está disponible, entonces falla (si todas las guardas fallan), o bien suspende (si alguna guarda suspende), a la espera de que otros agentes que estén actuando concurrentemente añadan información al almacén. La composición paralela está representada por $\mid\mid$. En caso de que todos los componentes de un sistema de agentes en paralelo estén suspendidos, se dice que el sistema está bloqueado. El agente $\exists_x A$ se comporta como A, donde x se toma como *local* a A. El agente *Stop* representa la terminación con éxito.

Por último, el agente $p(\tilde{x})$ representa la invocación de un procedimiento, donde p es el nombre del procedimiento y $\tilde{x}$ la lista de parámetros reales aplicados en la llamada. El significado de $p(\tilde{x})$ viene dado por una declaración de la forma $p(\tilde{y})$:- A, donde $\tilde{y}$ es la lista de parámetros formales. Se asume que hay exactamente una declaración por cada uno de los nombres de procedimiento que aparecen en una especificación.

La sencillez del modelo de coordinación que acabamos de describir ha facilitado su implementación en gran variedad de entornos y en combinación con numerosos lenguajes computa-

cionales, ya sean imperativos, lógicos, u orientados a objetos. Como ejemplo, la Figura 1.15 muestra cómo se implementa la multiplicación de matrices siguiendo un esquema *master/slave* en C-Linda, es decir, utilizando C para implementar las computaciones y Linda para los aspectos de coordinación.

```
void matmult() {
  int rows, cols, size, i, j;
  rd(‘‘A’’, ‘‘size’’, ?rows, ?size);
  rd(‘‘B’’, ‘‘size’’, size, ?cols);
  for (i=0; i<rows; i++)
    for(j=0; j<cols; j++)
      eval(‘‘C’’, i, j, calc(i,j,size));
  out(‘‘C’’, ‘‘size’’, rows, cols);
  }

double calc(int row, int col, int size) {
  double a, b, c = 0;
  int k;
  for (k=0; k <size; k++) {
    rd(‘‘A’’, row, k, ?a);
    rd(‘‘B’’, k, col, ?b);
    c += a*b;
    }
  return c;
  }
```

Figura 1.15: Multiplicación de matrices en C-Linda.

La rutina `matmult` se encarga de la multiplicación de dos matrices `‘‘A’’` y `‘‘B’’`, para lo que comienza leyendo las tuplas que indican el tamaño (`‘‘size’’`) de ambas matrices. Estas operaciones de lectura asignan valores a las variables `rows`, `size` y `cols`. A continuación, para cada uno de los elementos de la matriz resultado `‘‘C’’`, se evalúa en paralelo una tupla activa que calcula el valor de dicho elemento, invocando para ello la rutina `calc`. El resultado de la evaluación (es decir, el elemento en cuestión) se añade al espacio de tuplas. Finalmente se añade una tupla con el tamaño de la matriz resultado `‘‘C’’`. Respecto a la rutina `calc`, encargada de calcular un elemento de la matriz resultado, se observa cómo realiza la lectura de las tuplas correspondientes a la fila y la columna a multiplicar, y devuelve el resultado de la multiplicación.

Lógica temporal de acciones

La lógica temporal de acciones (TLA) es un lenguaje para describir el comportamiento de programas concurrentes y razonar sobre ellos basado en la notación matemática habitual y en la lógica temporal [Lamport, 1991].

TLA añade a la notación matemática clásica tres operadores: $'$, $\Box$ y $\exists\!\!\exists$. El primero de ellos permite distinguir entre los valores de las variables antes y después de la ejecución de una expresión, mientras que $\Box$ es el operador *siempre* de la lógica temporal clásica y $\exists\!\!\exists$ es un cuantificador temporal, que se diferencia del cuantificador existencial $\exists$ en que este último

afirma la existencia de un solo valor, mientras que $\exists$ afirma la existencia de una serie de valores —uno para cada estado de un comportamiento. Por lo demás, $\exists$ obedece a las reglas habituales del cálculo de predicados para la cuantificación existencial.

Una especificación en TLA es una fórmula lógica Φ que describe todos los posibles comportamientos correctos de un sistema. Esta fórmula consta de tres partes: una condición inicial *Init* que debe cumplir el estado inicial del programa, una relación N que establece las acciones posibles del programa como transiciones entre estados, y un conjunto de fórmulas L que establecen la *imparcialidad* del programa, de manera que tarde o temprano las acciones que estén habilitadas se ejecutarán:

$$\Phi = Init \wedge \Box[\mathcal{N}]_v \wedge L$$

donde v es un conjunto de variables relevantes a cada uno de las acciones del conjunto N.

Las fórmulas TLA permiten razonar sobre *comportamientos*, donde un comportamiento es una secuencia infinita de estados $\langle s_0, s_1, s_2, \ldots \rangle$ y un estado es una función que asigna valores a variables. Dada una especificación Φ en TLA y un comportamiento σ, es posible determinar si el comportamiento satisface la especificación. Para ello, debemos comprobar si el estado inicial s_0 satisface *Init*, si cada par $(s_i, s_{i+1}) \in \sigma$ bien satisface $\mathcal{N}$ o bien no modifica v, y si se satisface L.

El problema fundamental de este método es la falta de conexión entre TLA y un lenguaje de implementación concreto, puesto que no se dispone de una metodología clara para derivar programas a partir de las especificaciones, ni tampoco para realizar ingeniería inversa, *elevando* una implementación al nivel de la especificación para comprobar su corrección.

Notación Z

Otro enfoque para construir especificaciones es el que parte de la teoría de conjuntos y construye sobre ella una notación capaz de representar y modelar aplicaciones de software. De esta forma, el punto de partida de estas notaciones son los *tipos abstractos matemáticos*, junto con la lógica de predicados para razonar sobre ellos. En este campo, el formalismo más representativo es la notación Z [Spivey, 1992].

Z es una notación formal, basada en la teoría de conjuntos de Zermelo (de ahí su nombre), que utiliza los conceptos básicos de esta teoría (conjuntos, relaciones, funciones y variables) para describir sistemas. La notación proporciona una serie de tipos básicos, a los que se pueden añadir otros definidos por el diseñador.

Sobre estos tipos abstractos, Z define los llamados *esquemas*, que permiten la construcción estructurada y modular de aplicaciones de software. En Z los esquemas sirven para modelar tanto los aspectos estáticos (por ejemplo, los estados que puede alcanzar un sistema o los invariantes que deben conservarse en todos los estados) como los dinámicos (por ejemplo, las operaciones que pueden realizarse sobre el sistema, las relaciones entre las entradas y las salidas o la definición de cuándo y cómo se producen los cambios de estado).

Los esquemas se representan mediante cajas divididas en dos partes. En la superior se realiza la declaración del nombre de variables y sus tipos, mientras que la parte inferior expresa, de forma declarativa, la relación entre las variables que forman el esquema. La Figura 1.16 especifica una cola de naturales acotada a un máximo de diez elementos.

Para representar operaciones también se utilizan esquemas. Por ejemplo, la Figura 1.17 muestra la especificación de la operación que inserta un elemento en la cola. Las variables de Z

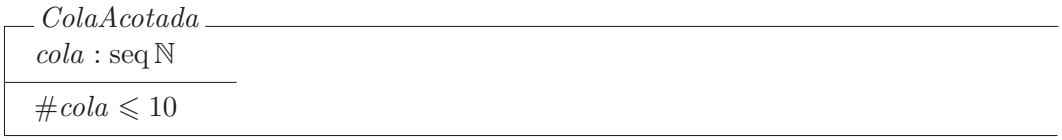


Figura 1.16: Especificación en Z del tipo abstracto *ColaAcotada*.

pueden ir *decoradas* con distintos símbolos. Así, '?' indica que la variable es un parámetro de entrada, mientras que '!' indicaría que se trata de un parámetro de salida, y '′' representa a la variable después de la ejecución de la operación. Las precondiciones se expresan también como relaciones y son aquéllas que no contienen variables decoradas (en este caso, $\#cola < 10$). Si una precondición no se satisface, la operación no se realiza.

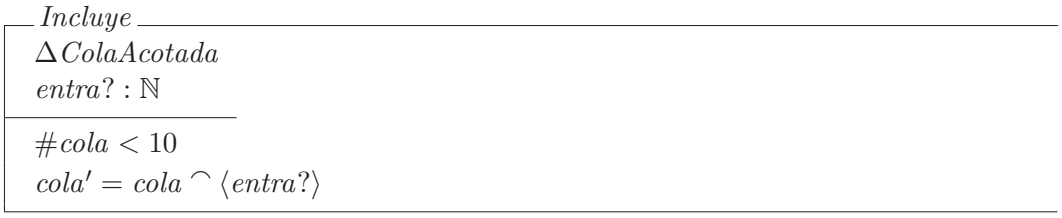


Figura 1.17: Especificación en Z de la operación *Incluye*.

La notación Z es muy expresiva para modelar entidades estáticas y su comportamiento, y entre sus ventajas destaca el que ofrezca una metodología para la derivación de implementaciones a partir de las especificaciones, mediante un proceso de refinamiento. Sin embargo, entre los inconvenientes debemos indicar la falta de estructura de las especificaciones, que son, por tanto, *planas*, en el sentido de que no existe posibilidad de realizar especificaciones modulares ni anidadas: todas las variables son globales. Además, Z no es adecuada para modelar sistemas reactivos, en los que es necesario utilizar conceptos como puede ser la concurrencia o características de tiempo real. Tampoco resulta sencillo describir aspectos como la evolución de la topología de un sistema dinámico. Aunque existen trabajos que muestran cómo vencer estas limitaciones, algunos de ellos lo hacen de forma un tanto forzada [Lamport, 1994]. Estas deficiencias han dado lugar a la propuesta de diversas extensiones de Z que permiten la construcción de especificaciones orientadas a objetos, como es el caso de Object-Z [Duke et al., 1994], que incorporan características de modularidad y concurrencia.

Lenguajes de especificación algebraica

Los lenguajes de especificación algebraica permiten realizar una descripción declarativa abstracta del comportamiento *correcto* que debe observar un programa. El carácter abstracto de la especificación viene dado por el hecho de ser independiente de todos aquellos detalles que sean irrelevantes para el usuario del sistema, en particular de cómo se implemente finalmente la especificación.

Desde este punto de vista, la especificación consiste en describir los sistemas por medio de las propiedades o *axiomas* que deben de satisfacer. Cada axioma se describe mediante una fórmula en alguna lógica, y la especificación total es la conjunción de todas las fórmulas. Diremos

que una implementación *satisface* una especificación si y sólo si es un modelo de la misma, es decir si hace válidos sus axiomas. En general, diremos que una especificación X *verifica* a otra especificación Y, si los axiomas de Y pueden deducirse como teoremas de los de X; en otras palabras, la verificación se reduce a implicaciones lógicas.

Formalmente, una especificación axiomática es un par (Σ, E), donde Σ es un alfabeto que define las operaciones que se pretenden especificar y E es un conjunto de axiomas que relacionan las operaciones de Σ. Asociado a la especificación existe un *álgebra de términos*, que contiene todas las fórmulas bien formadas obtenidas a partir de las operaciones del alfabeto E. Para facilitar el razonamiento sobre las especificaciones, normalmente se impone la condición de que los axiomas sean ecuaciones que igualan términos de dicho álgebra, en cuyo caso la especificación se denomina *algebraica*. Representaremos como $T_{\Sigma,E}(X)$ a la Σ-álgebra de clases de equivalencia de términos de Σ módulo las fórmulas del conjunto de axiomas E y con variables en un conjunto dado $X = \{x_1, x_2, \ldots, x_n\}$.

Este estilo de especificación basado en ecuaciones presenta dos utilidades principales. En primer lugar, si interpretamos los axiomas como reglas de reescritura de izquierda a derecha, es posible construir prototipos de los sistemas especificados. Estos prototipos se basan en la reducción de términos hasta llegar a su forma normal, que es la expresión más simple de los representantes canónicos de cada clase de equivalencia de $T_{\Sigma,E}(X)$. En segundo lugar, es posible razonar sobre los sistemas especificados demostrando propiedades sobre ellos a partir de sus axiomas. Para demostrar una propiedad P sobre un objeto del sistema dado por un término t de tipo T, basta con especificar P como una operación $P : T \rightarrow Bool$ y reducir el término $P(t)$. Otros métodos de razonamiento disponibles son la inducción estructural y el razonamiento ecuacional.

Existen diversos lenguajes de especificación algebraica, entre los que podemos citar Troll [Hartel et al., 1997], OBJ [Goguen et al., 2000] y Maude [Clavel et al., 1999, Clavel et al., 2000]. La Figura 1.18 muestra el estilo de estos lenguajes al representar la especificación algebraica de una lista genérica.

La especificación comienza definiendo una teoría *TRIVIAL* que se utiliza a continuación como parámetro formal de la lista genérica. La teoría representa los requisitos que debe cumplir cualquier módulo con el que se parametrice la lista. En este caso, basta con que el módulo defina un dominio, al que se hace referencia con el nombre *Item*.

Seguidamente la especificación declara dos dominios parametrizados: $Lista[X]$ y $ListaNV[X]$, donde este último representa a las listas no vacías. La declaración de subdominios a continuación indica que el dominio *Item* es un subconjunto de las listas no vacías (concretamente el subconjunto que representa a las listas con un sólo elemento), y que las listas no vacías son, a su vez, un subconjunto de las listas. Eso último nos va a servir especificar operaciones parciales (como *cabeza* y *cola*) que sólo están definidas sobre las listas no vacías.

A continuación se define la signatura de las operaciones del dominio, incluyendo los constructores para representar la lista vacía (*nula*) y la operación de concatenación de listas (que se define como asociativa y con elemento neutro *nula*) y el resto de las operaciones del módulo (*cabeza*, *cola* y *vacía*), donde *Bool* es un dominio predefinido que especifica los valores lógicos. Finalmente se indican los axiomas o ecuaciones que sirven para indicar la semántica de las operaciones al relacionar unas con otras, a la vez que pueden ser utilizadas para la reducción de términos.

Podemos utilizar el módulo *MACHINE-INT*, que especifica los enteros, para instanciar el parámetro formal de la lista. De esta forma obtenemos un módulo $LISTA[MACHINE\text{-}INT]$ que especifica las listas de enteros. Ahora podemos reducir un término de este dominio utilizando

```
fth TRIVIAL is
  sort Item .
endfth

fmod LISTA[X :: TRIVIAL] is
   sorts Lista[X] ListaNV[X] .
   subsorts Item < ListaNV[X] < Lista[X] .
   op nula : -> Lista[X] .
   op __ : Lista[X] Lista[X] -> Lista[X] [assoc id : nula] .
   op __ : ListaNV[X] Lista[X] -> ListaNV[X] [assoc] .
   op cabeza : ListaNV[X] -> Item .
   op cola : ListaNV[X] -> Lista[X] .
   op vacía : Lista[X] -> Bool .
   var I : Item .
   var L : Lista .
   eq cabeza(I L)) = I .
   eq cola(I L) = L .
   eq vacía(L) = L == nula .
endfm
```

Figura 1.18: Especificación en Maude de una lista genérica.

para ello los axiomas de la especificación. Por ejemplo, la expresión:

```
reduce in LISTA[MACHINE-INT] : 4 cola(cola( 3 2 1 nula))
```

tendría como resultado, mediante la aplicación repetida de los axiomas como reglas de reescritura, el término *4 1*, forma canónica de la clase de equivalencia que representa la lista que contiene los enteros 4 y 1.

Frente a las ventajas ya citadas de demostración de propiedades y posibilidad de ejecutar las especificaciones, los inconvenientes que presentan las especificaciones algebraicas residen fundamentalmente en su bajo nivel, que hace que la especificación de cualquier sistema no trivial se convierta en difícilmente manejable, debido a su extensión. Por otro lado, el carácter estrictamente funcional de este tipo de especificaciones dificulta notablemente la descripción de los efectos laterales que, con frecuencia, implica la realización de una operación, como pueden ser, por ejemplo, los producidos por operaciones de E/S.

1.6 Requisitos de un lenguaje de descripción de arquitectura

A lo largo de este capítulo hemos ofrecido una visión general de lo que es la Arquitectura del Software, su objeto de estudio y las principales líneas de investigación actualmente existentes en este campo, así como de dónde se encuadra esta disciplina respecto a otras tendencias recientes en Ingeniería del Software. Igualmente, hemos realizado un estudio de los distintos ADLs existentes y de los diversos formalismos utilizados en Arquitectura del Software.

Este estudio muestra que se trata de una disciplina en la que se están dando aún los primeros pasos, en la que no existe todavía un consenso definitivo sobre cuáles son los aspectos de interés

arquitectónico, ni sobre cómo abordarlos debidamente, por lo que existe una gran diversidad de propuestas y aproximaciones al problema.

Llegados a este punto, y como conclusión de este capítulo introductorio, trataremos de realizar una puesta en común, desde un punto de vista crítico, de estas aproximaciones a los aspectos arquitectónicos del software, determinando las características principales que un ADL debería ofrecer para ser útil como vehículo de representación de la arquitectura de los sistemas de software. Esto nos sirve para establecer los requisitos que guian el diseño de LEDA, el lenguaje de descripción de arquitectura sobre el que trata este trabajo y que se describe de forma detallada en el Capítulo 3. Hemos agrupado estos requisitos bajo distintos epígrafes, que tratan sobre las entidades que maneja el lenguaje, la descripción de arquitecturas dinámicas, la verificación de propiedades, el desarrollo del sistema, la generación de código y la evolución de la arquitectura.

Componentes y conectores

En primer lugar, es necesario determinar las entidades que va a manejar el lenguaje, es decir, cuáles son los bloques de construcción básicos que, combinados de forma adecuada, permitan describir la arquitectura de un sistema de software.

A este respecto, resulta conveniente que prestemos atención a los conceptos de *componente* y *conector* en el sentido en que son utilizados en la literatura sobre Arquitectura del Software. En este contexto, los componentes son unidades computacionales o de almacenamiento de datos, mientras que los conectores modelan las interacciones entre componentes y las reglas que rigen dichas interacciones. Ejemplos de componentes serían, por tanto, módulos computacionales, ficheros, bases de datos, procesos, filtros, etc., es decir, elementos de software claramente identificables como piezas concretas de un sistema, tanto a nivel de su representación abstracta (por ejemplo, estarían presentes como tales elementos diferenciados en los diagramas utilizados en la fase de especificación), como en su implementación final. Por otro lado, ejemplos de conectores serían elementos más intangibles dentro de un sistema de software, tales como entrada/salida mediante ficheros, acceso a variables, llamadas a procedimiento (locales o remotas), etc. En la mayoría de los casos, estos elementos no pueden identificarse de forma separada en el sistema de software una vez implementado, aunque sí pueden aparecer, de forma distinta a los componentes, en los diagramas previos utilizados durante el desarrollo [Shaw y Garlan, 1996].

Resulta evidente que un requisito fundamental a tener en cuenta en el diseño de un ADL es que incorpore el concepto de componente, considerado como una caja negra que implementa una funcionalidad determinada y presenta una interfaz que establece las necesidades y posibilidades de conexión de unos componentes con otros. Esta noción de componente como unidad básica para la construcción (y reutilización) de sistemas de software es compartida por todos los ADLs a los que hemos hecho referencia en la Sección 1.4.5, y entronca también con los principios de la Ingeniería del Software Basada en Componentes (CBSE), a la que hemos dedicado la Sección 1.1.

Sin embargo, no resulta tan clara la necesidad de considerar también a los conectores como entidades del lenguaje [Luckham et al., 2000]. En efecto, tal como hemos visto algunos ADLs, como Wright o Unicon, les dan esta categoría de entidades de primera clase, mientras que otros los relegan a un segundo plano, como es el caso de C2, y en algunos más, por ejemplo en Rapide o Darwin, simplemente no existen como tales.

Desde nuestro punto de vista, la distinción entre componentes y conectores es, por lo general, demasiado sutil. Con frecuencia los artefactos de software comparten características tanto de componentes (en el sentido que realizan una determinada computación) como de conectores (al

servir para conectar otros componentes entre sí). Por otro lado, la composición de componentes y conectores llevaría a la creación de compuestos *híbridos* que ya no podrían ser clasificados atendiendo a esta dicotomía componente/conector.

Por los motivos que acabamos de exponer, y con objeto de mantener una cierta regularidad y simplicidad conceptual, consideraremos ambas categorías —componentes y conectores— genéricamente como componentes. De esta forma, no se establece un compromiso con un modelo arquitectónico determinado ni se imponen mecanismos de interacción concretos entre los componentes. No quiere esto decir que no seamos conscientes de la importancia de los conectores, sino simplemente que no consideramos necesario darles un tratamiento específico y diferenciado: todos los elementos de interés arquitectónico se pueden modelar como tipos específicos de componentes.

Dinamismo

El objetivo de hacer explícitos los aspectos arquitectónicos del software es facilitar el desarrollo y evolución de sistemas complejos. Debido al carácter inherentemente dinámico y reconfigurable de muchos de estos sistemas, la capacidad de describir los aspectos que rigen la evolución de una arquitectura es un requisito básico de cualquier ADL [Medvidovic y Rosenblum, 1997]. Las arquitecturas dinámicas admiten la replicación, inserción, eliminación y reconexión de sus componentes en tiempo de ejecución.

Sin embargo, debemos tener en cuenta que un excesivo dinamismo está reñido necesariamente con la descripción de la arquitectura [Magee y Kramer, 1996], por lo que es necesario llegar a un compromiso en este aspecto. En efecto, como ejemplo de sistema totalmente dinámico podríamos considerar uno formado simplemente por una colección de componentes que 'nadan' en algún medio que permita la transmisión de mensajes o la invocación de operaciones entre ellos. Para integrar un nuevo componente en el sistema bastaría con introducirlo en dicho medio, y serían los propios componentes los encargados de establecer y reconfigurar sus relaciones en función de las interfaces que unos y otros ofrezcan. Este ejemplo puede considerarse como paradigmático de los sistemas abiertos, siendo muy utilizado para modelar este tipo de sistemas [Vallecillo, 1999], pero en él no es posible especificar nada acerca de la estructura del sistema, ni sobre cuáles son los componentes que lo forman, o qué papeles juegan unos en relación con otros.

En el extremo opuesto, podemos considerar una arquitectura rígida, que establezca de forma estricta cuáles son sus componentes y cómo interactúan unos con otros, de forma que la descripción de la arquitectura sea completamente conocida ya durante la fase de diseño, pero que no permita ninguna flexibilidad que pueda hacer evolucionar el sistema en el tiempo.

Por tanto, un ADL debe aportar mecanismos de creación de componentes y de reconfiguración dinámica del sistema, de manera que se dote de flexibilidad a la arquitectura en tiempo de ejecución, pero sin perder por ello las ventajas que aporta la descripción estructural durante la especificación del sistema.

De entre los ADLs estudiados en la Sección 1.4.5, sólo Darwin aborda de forma efectiva la descripción de arquitecturas dinámicas. C2 y Rapide permiten un cierto grado de dinamismo, pero que está limitado en el primero de ellos por las restricciones del estilo arquitectónico en el que se basa C2, mientras que en Rapide las posibilidades se reducen a la descripción de una serie de configuraciones alternativas que puede presentar el sistema. Los otros dos lenguajes considerados —UniCon y Wright— sólo permiten la descripción estática de la arquitectura.

Verificación de propiedades

El éxito del diseño arquitectónico depende en gran medida de que se logre realizar especificaciones precisas de los sistemas, sus componentes y las interacciones entre ellos. Los requisitos referidos al análisis abordan la necesidad de servir de apoyo al razonamiento, tanto automatizado como no automatizado, acerca de las descripciones arquitectónicas. Estas descripciones modelan a menudo complejos sistemas de software concurrentes o distribuidos. La comprobación de propiedades de dichos sistemas en las etapas iniciales del desarrollo, como es la de diseño arquitectónico, reducen substancialmente los costes de los errores [Allen y Garlan, 1992].

Con objeto de realizar dichos análisis, la notación utilizada debe estar basada en un formalismo que le sirva de soporte. Un modelo formal proporciona una definición precisa de una arquitectura de software. La especificación de una arquitectura debe cubrir una amplia gama de propiedades, de forma que fuese posible realizar ricos y variados análisis a partir de las descripciones arquitectónicas. Sin embargo, los tipos de análisis para los que un ADL es adecuado dependen del modelo semántico subyacente, con lo que los formalismos que resultan adecuados para el análisis de propiedades dinámicas de la arquitectura pueden no serlo tanto para realizar análisis estáticos o de propiedades globales [Shaw y Garlan, 1995]. Por ejemplo, Wright (que está basado en CSP) permite realizar análisis estáticos acerca de la compatibilidad de las conexiones y la ausencia de bloqueos, pero no permite la creación dinámica de procesos. Por el contrario, Darwin (basado en el cálculo π), ofrece una gran flexibilidad a la hora de describir propiedades dinámicas, pero es mucho más limitado a la hora de realizar comprobaciones de tipos. En cualquier caso, parece claro que las propiedades más interesantes son las relacionadas con la definición de la forma en la que los componentes interactúan, que determina, por tanto, de qué forma pueden ser combinados unos con otros para formar sistemas [Allen y Garlan, 1997].

La elección de un formalismo base adecuado determinará en gran medida las posibilidades de análisis que vaya a tener el ADL que sobre él se diseñe. A la hora de tomar una decisión sobre el formalismo a utilizar, también será un aspecto muy a tener en cuenta la disponibilidad de herramientas de análisis (o la mayor o menor dificultad para construirlas) que permitan sacar partido de la especificación arquitectónica.

Como resumen, diremos que ni UniCon ni C2 contemplan la verificación de propiedades de los sistemas descritos. Por su parte, Wright permite analizar la compatibilidad de la conexiones entre componentes y roles, mientras que Darwin permite el análisis de diversas propiedades de seguridad y viveza. El análisis de propiedades en Rapide consiste en la verificación del cumplimiento de restricciones del comportamiento de los componentes que pueden ser expresadas en la arquitectura.

Desarrollo del sistema y reutilización

El argumento más utilizado para el desarrollo y uso de modelos y descripciones arquitectónicas del software es que son necesarios para cubrir el hueco existente entre los diagramas informales de cajas y líneas, como a los descritos en la Sección 1.4.1, y la implementación final de dichos sistemas.

El proceso de desarrollo de cualquier sistema de software de cierta complejidad pasa normalmente por una serie de refinamientos sucesivos, en los que el sistema se representa a diferentes niveles de abstracción que lo van acercando de manera progresiva desde la especificación a la implementación. Esto es lo que se conoce con el nombre de desarrollo iterativo e incremental, y es la base de las propuestas más recientes de modelos de ciclo de vida del software y del proceso de desarrollo [Jacobson et al., 1999]. Un buen ADL debería proporcionar mecanismos

para el refinamiento de la arquitectura y sus componentes que faciliten este proceso de desarrollo [Medvidovic y Taylor, 2000]. Además, es deseable poder manejar especificaciones parciales e incompletas, que se vayan refinando progresivamente.

De este modo, una especificación arquitectónica de alto nivel puede utilizarse como medio de comprensión y comunicación de un sistema en las etapas iniciales de su desarrollo. Posteriormente, una descripción que incorpore mayor grado de detalle permitirá realizar un análisis de la consistencia de las conexiones entre sus componentes. Aumentar aún más el grado de detalle puede servir para realizar una simulación o prototipado del sistema, para lo que debería de contemplarse la generación de código a partir de la especificación arquitectónica. Incluso la etapa de implementación del sistema no tiene por qué plantearse desde un punto de vista de todo o nada, en el que se cubra de un salto la distancia entre la especificación y el sistema ya implementado. Por el contrario, es deseable la posibilidad de manejar descripciones arquitectónicas en las que parte de los componentes estén implementados y parte no, permaneciendo estos últimos especificados en la notación del ADL. Este proceso de refinamiento debe hacerse de forma que se preserven, en los distintos niveles de abstracción, las propiedades del sistema verificadas durante el análisis.

Por otro lado, una vez finalizado el proceso de desarrollo, es necesario servir de apoyo a la evolución del software. Esto constituye un aspecto clave de cualquier notación, método o herramienta de desarrollo, también por tanto para la Arquitectura del Software. La arquitectura evoluciona para reflejar la evolución de un sistema de software concreto, y también para dar lugar a familias de sistemas relacionados. Esta posibilidad de hacer evolucionar el diseño arquitectónico es precisamente la que permite su reutilización en diferentes contextos. Sin embargo, la reutilización efectiva de una arquitectura requiere a menudo que alguno de sus componentes sea eliminado, reemplazado o reconfigurado sin afectar por ello a otras partes de la aplicación [Nierstrasz y Meijler, 1995].

Por su parte, los componentes individuales de una arquitectura también pueden evolucionar. Si consideramos a los componentes como tipos que se instancian cada vez que se usan en una arquitectura, su evolución puede considerarse en términos de subtipado o especialización de tipos [Nierstrasz, 1995a]. De esta forma, podremos reemplazar un componente en un arquitectura por otro que sea un versión especializada del primero. Esto da lugar a un mecanismo de instanciación mediante el cual una arquitectura de software puede considerarse como un marco de trabajo genérico que puede ser parametrizado parcialmente y reutilizado tantas veces como sea necesario.

Resulta, por tanto, evidente la necesidad de que los ADLs contemplen mecanismos para el refinamiento y evolución de componentes y arquitectura, mecanismos que han de basarse en la especialización y parametrización, de forma que se facilite tanto el proceso de desarrollo en sí, como la posterior evolución del sistema y la reutilización del diseño arquitectónico. Del mismo modo, un valor añadido a tener en cuenta en cualquier ADL será la disponibilidad de herramientas para la simulación de las especificaciones y la automatización de la implementación del sistema final a partir de las mismas.

Sin embargo, la mayoría de los lenguajes estudiados en la Sección 1.4.5 carecen de mecanismos que faciliten el refinamiento y la reutilización. Sólo Rapide presenta un mecanismo de herencia para la adaptación de componentes, mientras que C2 establece diversas formas de subtipado. Con respecto a la ejecución de las especificaciones, debemos decir que Wright es el único de los lenguajes que no contempla estos aspectos, mientras que el resto dispone de herramientas de generación de código. Además, Rapide permite la simulación de las arquitecturas descritas en este lenguaje.

A modo de resumen final, la Figura 1.19 muestra una tabla comparativa de diferentes ADLs respecto a las características o requisitos que acabamos de enunciar. En el Capítulo 5, dedicado a las conclusiones de este trabajo, se muestra de nuevo esta tabla, pero incluyendo a LEDA entre los lenguajes comparados en ella.

	Entidades	Dinamismo	Verificación de propiedades	Desarrollo y reutilización	Ejecución
UniCon	Componentes y conectores	No	No	No	Generación de código
Wright	Componentes y conectores	No	Compatibilidad	No	No
Darwin	Componentes	Sí	Seguridad y viveza	No	Generación de código
Rapide	Componentes	Limitado	Análisis de restricciones	Herencia	Simulación y generación
C2	Componentes	Limitado	No	Subtipado	Generación de código

Figura 1.19: Tabla comparativa de las características de los ADLs estudiados.

El próximo capítulo está dedicado a la base formal de la que hemos dotado a nuestro lenguaje de descripción de arquitectura, y más concretamente a las relaciones de compatibilidad y herencia de comportamiento, definidas en el contexto del cálculo π, que permiten el análisis de las especificaciones escritas en LEDA.

Capítulo 2

Fundamentos formales

La aplicación de métodos formales al desarrollo del software depende de la disponibilidad de modelos y formalismos adecuados para cada una de las etapas del proceso de desarrollo. Este trabajo se enmarca dentro del nivel de diseño de aplicaciones denominado Arquitectura del Software, en el que el sistema se describe mediante una colección de componentes interrelacionados. Nuestro enfoque se basa en el uso de un álgebra de procesos como soporte formal para la descripción y análisis de la arquitectura del software.

Desde el punto de vista de las álgebras de procesos, y tal como indicamos en la Sección 1.5.1, un sistema distribuido es visto como un proceso, probablemente consistente en la composición de varios subprocesos, lo que lleva a que la especificación describa el sistema a través de una jerarquía de definiciones de procesos. Los procesos son entidades capaces de llevar a cabo acciones internas (no observables) y acciones de interacción con su entorno. Las interacciones consisten en acciones elementales de sincronización, por lo general con intercambio de datos.

De entre las diversas álgebras de procesos existentes, hemos elegido el cálculo π como base formal de nuestro trabajo debido fundamentalmente a la naturalidad con la que se pueden describir sistemas dinámicos utilizando este formalismo, y también a las posibilidades que ofrece para representar los cambios de estado en sistemas orientados a objetos. A diferencia de otras álgebras de procesos, como CCS o CSP, el cálculo π puede expresar la *movilidad* de forma directa, mediante la transmisión de referencias a procesos —o enlaces— que se envían como argumentos en la comunicación entre procesos. Esto lo hace especialmente adecuado para describir la estructura de sistemas de software en los que los componentes puedan ser creados y eliminados de forma dinámica, y en los que las conexiones entre componentes se puedan establecer y modificar también dinámicamente, haciendo que la topología de comunicación del sistema no sea rígida, sino que evolucione en el tiempo. Diremos que estos sistemas presentan una *arquitectura dinámica*. Ejemplos típicos de esta clase de aplicaciones son los sistemas abiertos y distribuidos. Las características especiales de estos sistemas dinámicos precisan de la realización de cambios en los métodos y herramientas de la Ingeniería del Software, con objeto del abordar los nuevos requisitos.

La estructura de este capítulo es la siguiente. En primer lugar figura una breve introducción al cálculo π que esperamos facilite al lector no familiarizado con este formalismo la comprensión del presente capítulo, así como del resto de esta memoria. A continuación mostramos la modelización en el cálculo π de un espacio de tuplas como el usado por algunos lenguajes de coordinación. El objeto de esta sección es mostrar la expresividad del cálculo así como la posibilidad de representar en él distintos modelos y estilos arquitectónicos. Procedemos a continuación a presentar nuestra propuesta para la especificación de la arquitectura del software,

incluyendo definiciones formales para las nociones intuitivas de rol, conexión y arquitectura habitualmente presentes en la literatura sobre Arquitectura del Software. A continuación definimos una relación de compatibilidad entre agentes en el cálculo π, a la vez que presentamos los resultados que permiten determinar la compatibilidad entre roles y asegurar el éxito de su conexión, así como la composicionalidad de una determinada arquitectura. En la sección siguiente, y con objeto de garantizar la reemplazabilidad de componentes (es decir, su sustitución por otros en una arquitectura, manteniendo la compatibilidad), presentamos sendas relaciones de herencia y extensión de comportamiento, así como demostramos que dichas relaciones preservan la compatibilidad. Es en estas dos últimas secciones donde figura el grueso de los desarrollos formales del presente trabajo. Para finalizar, realizamos un repaso de los trabajos más estrechamente relacionados con el nuestro y presentamos algunas conclusiones. A lo largo de todo el capítulo, un gran número de ejemplos pretende iluminar las cuestiones que se abordan en él, así como mostrar la aplicación de nuestras propuestas.

2.1 El cálculo π

El cálculo π [Milner et al., 1992], desarrollado por Milner y sus colaboradores a partir de CCS, permite expresar de forma directa la *movilidad* [Engberg y Nielsen, 1986], lo que le hace especialmente indicado para la descripción y análisis de sistemas concurrentes cuya topología sea dinámica. Estos sistemas se representan en el cálculo π mediante colecciones de procesos o *agentes* que interactúan por medio de enlaces o *nombres*. Dichos nombres pueden considerarse como canales bidireccionales compartidos por dos o más agentes. La restricción del ámbito de un enlace permite el establecimiento de enlaces privados a un grupo de agentes.

El cálculo π se diferencia de otras álgebras de proceso en que no distingue entre enlaces y datos, siendo ambos considerados genéricamente como nombres. Este tratamiento homogéneo de los nombres permite construir un cálculo muy simple, pero también muy potente; lo que se envía a través de los nombres de enlaces son, a su vez, nombres de enlaces. Así, la acción $\bar{x}y$ representa la emisión del nombre de enlace y a través del nombre de enlace x. En una acción de esta forma, diremos que x actúa como *sujeto* de la comunicación, mientras que y actúa como *objeto*. Es precisamente esta regularidad a la hora de tratar los nombres la que permite expresar de forma directa la movilidad en el cálculo π. Cuando un agente recibe un nombre, como objeto o argumento en una comunicación a través de un enlace, puede usar este nombre como sujeto para una transmisión futura, lo que resulta en una forma sencilla y efectiva de reconfigurar el sistema.

2.1.1 Sintaxis

Sea $(P, Q \in)\mathcal{P}$ el conjunto de agentes, y $(w, x, y \in)\mathcal{N}$ el de nombres. Las secuencias de nombres se abrevian normalmente utilizando tildes ($\tilde{w}$). Según esto, los agentes se definen recursivamente a partir de nombres y agentes como sigue:

$$\mathbf{0} \mid (x)P \mid [x = z]P \mid \tau.P \mid \bar{x}y.P \mid \bar{x}(y)P \mid x(w).P \mid P \mid Q \mid P + Q \mid \; ! \, P \mid A(\tilde{w})$$

- El comportamiento vacío o inactivo se denota mediante la *inacción*, representada como $\mathbf{0}$.
- Las restricciones se utilizan para crear nombres privados. Así, en $(x)P$, el nombre x denota un enlace privado de P, de forma que la comunicación entre P y cualquier otro agente

utilizando x como sujeto, está prohibida. Sin embargo, sí se permite esta comunicación a través de x *en el interior* de P, es decir, entre sus componentes. El ámbito de un nombre puede ser ampliado mediante el envío de este nombre a otro agente (véase la acción de *salida ligada* más abajo).

- Un agente con guarda (*match*), descrito como $[x = z]P$, se comporta como P si los nombres x y z son idénticos, y como $\mathbf{0}$ en caso contrario. Las guardas son innecesarias en el cálculo π para escribir computaciones sobre tipos de datos, puesto que su funcionalidad puede ser expresada mediante otros métodos. Sin embargo, se suelen utilizar con objeto de obtener codificaciones más sencillas.

- Las transiciones silenciosas, representadas mediante τ, modelan acciones internas. Así, un agente $\tau.P$ evolucionará en algún momento a P sin interactuar con su entorno.

- Un agente prefijado por una acción de salida, como $\bar{x}y.P$, envía el nombre y (objeto) a través del nombre x (sujeto), comportándose después como P.

- Un agente prefijado por una acción de entrada, como $x(w).P$, espera a que un nombre y sea enviado a través de x, y a continuación se comporta como $P\{y/w\}$, donde $\{y/w\}$ es la sustitución de w por y.

- Además de estas tres acciones básicas, existe también una derivada —la salida ligada, expresada mediante $\bar{x}(y)$—, que representa la emisión a través de un enlace x de un nombre y *privado* del agente que realiza dicha emisión, ampliando de esta forma el ámbito de este nombre. La salida ligada es simplemente una abreviatura de $(y)\bar{x}y$, pero debe ser considerada de forma separada puesto que está sometida a reglas de transición ligeramente diferentes de las que rigen las acciones de salida libres.

- El operador de composición se define de la forma esperada: $P \mid Q$ se compone de P y Q actuando en paralelo.

- El operador de suma se utiliza para especificar alternativas: $P + Q$ puede progresar a P o a Q. La elección entre una y otra alternativa puede tomarse de forma local o global. En una *decisión global*, dos agentes se comprometen de forma síncrona en realizar acciones complementarias, como en:

$$(\cdots + \bar{x}y.P + \cdots) \mid (\cdots + x(w).Q + \cdots) \xrightarrow{\tau} P \mid Q\{y/w\}$$

  Por otro lado, se denominan *decisiones locales* a aquellas situaciones en las que un agente elige una alternativa entre varias con independencia de su contexto. Tal es el caso de un agente como:

$$(\cdots + \tau.P_1 + \tau.P_2 + \cdots)$$

  que puede progresar a P_1 o a P_2 (realizando en ambos casos una acción silenciosa τ) con independencia de cualquier proceso que forme parte de su contexto. Utilizaremos decisiones locales y globales para indicar las responsabilidades de acción y reacción.

  Tanto el operador de composición como el de suma pueden aplicarse a un conjunto finito de agentes $\{P_i\}_i$. En este caso, se representan como $\prod_i P_i$ y $\sum_i P_i$, respectivamente[1].

[1]Debido a que en el cálculo π los términos son finitos, nos referiremos exclusivamente a conjuntos finitos de agentes, prescindiendo por lo general de indicar rangos de índices en conjuntos, productos y sumas de agentes, Se entiende que dichos rangos son $1 \,..\, n$

$$\text{TAU: } \frac{-}{\tau.P \xrightarrow{\tau} P} \qquad \text{IN: } \frac{-}{\bar{x}y.P \xrightarrow{\bar{x}y} P} \qquad \text{OUT: } \frac{-}{x(z).P \xrightarrow{x(w)} P\{w/z\}}\ w \notin fn((z)P)$$

$$\text{MATCH: } \frac{P \xrightarrow{\alpha} P'}{[x=x]P \xrightarrow{\alpha} P'} \qquad \text{IDE: } \frac{P\{\tilde{y}/\tilde{x}\} \xrightarrow{\alpha} P'}{A(\tilde{y}) \xrightarrow{\alpha} P'}\ A(\tilde{x}) \stackrel{\text{def}}{=} P$$

$$\text{SUM: } \frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'} \qquad \text{PAR: } \frac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q}\ bn(\alpha) \cap fn(Q) = \varnothing$$

$$\text{COM: } \frac{P \xrightarrow{\bar{x}y} P' \quad Q \xrightarrow{x(z)} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'\{y/z\}} \qquad \text{CLOSE: } \frac{P \xrightarrow{\bar{x}(w)} P' \quad Q \xrightarrow{x(w)} Q'}{P \mid Q \xrightarrow{\tau} (w)(P' \mid Q')}$$

$$\text{RES: } \frac{P \xrightarrow{\alpha} P'}{(y)P \xrightarrow{\alpha} (y)P'}\ y \notin n(\alpha) \qquad \text{OPEN: } \frac{P \xrightarrow{\bar{x}y} P'}{(y)P \xrightarrow{\bar{x}(w)} P'\{w/y\}}\ y \neq x \wedge w \notin fn((y)P')$$

Figura 2.1: Sistema de transiciones del cálculo π.

- El operador de replicación (**!** , que se lee como *bang*) es útil para representar la iteración y la recursión, aunque se trata de un operador derivado. El proceso $!\,P$ representa la replicación ilimitada del proceso P, es decir, $!\,P$ puede producir tantas réplicas en paralelo de P como sea necesario, y es equivalente a $P \mid\ !\,P$. No hay reglas de reducción para $!\,P$ y, en concreto, el término P no puede comenzar a reducirse hasta que sea expandido como $P \mid\ !\,P$.

- Por último, $A(\tilde{w})$ es un agente definido. Cada identificador de agente A se define mediante una única ecuación $A(\tilde{w}) = P$. El uso de identificadores de agentes permite la definición modular y recursiva de agentes.

El conjunto de nombres de un agente P se denota mediante $n(P)$. Los *nombres libres* de P, $fn(P)$, son aquellos nombres en $n(P)$ que no están ligados por una acción de entrada o una restricción. Denotamos mediante $bn(P)$ a los *nombres ligados* de P.

2.1.2 Sistema de transiciones

El sistema de transiciones utilizado en este trabajo es el propuesto en [Milner et al., 1992], y se muestra en la Figura 2.1. Las reglas relativas a los operadores binarios + y | tienen formas simétricas adicionales. Como se puede observar en dicha figura, las transiciones se representan mediante flechas etiquetadas. Así, $P \xrightarrow{\alpha} P'$ indica que el proceso P realiza una acción α convirtiéndose entonces en P'. Aparte de esta transición básica, utilizaremos las siguientes notaciones abreviadas:

- $\Longrightarrow$ representa $(\xrightarrow{\tau})^*$, el cierre reflexivo y transitivo de $\xrightarrow{\tau}$.
- $\stackrel{\alpha}{\Longrightarrow}$ representa $\Longrightarrow\xrightarrow{\alpha}\Longrightarrow$ siempre que $\alpha \neq \tau$.
- $P \xrightarrow{\alpha}$ representa $\exists\, P'\,.\, P \xrightarrow{\alpha} P'$.

- $P \overset{\alpha}{\not\longrightarrow}$ representa $\nexists P' \,.\, P \xrightarrow{\alpha} P'$.
- $P \Longrightarrow \mathbf{0}$ representa $\exists\ \mathbf{0}_P \,.\, P \Longrightarrow \mathbf{0}_P$ y $\mathbf{0}_P \equiv \mathbf{0}$.

2.1.3 Movilidad en el cálculo π

La expresión directa de la movilidad es una característica del cálculo π que lo distingue de otras álgebras de procesos como CSP o CCS. En el cálculo π, la movilidad se consigue mediante la transmisión de nombres de enlaces. Así, un agente puede enviar un enlace como objeto de una acción de salida. El agente que realice la acción de entrada complementaria logrará acceso a dicho enlace, pudiendo a partir de ese momento utilizarlo, como sujeto, para realizar acciones de entrada y de salida a través de él. Para ilustrar este punto, consideremos el siguiente ejemplo.

Ejemplo 2.1 (Transmisión de enlaces) *Supongamos que, tal como muestra la Figura 2.2 (izquierda), tenemos tres agentes: P, Q y R, combinados en paralelo.*

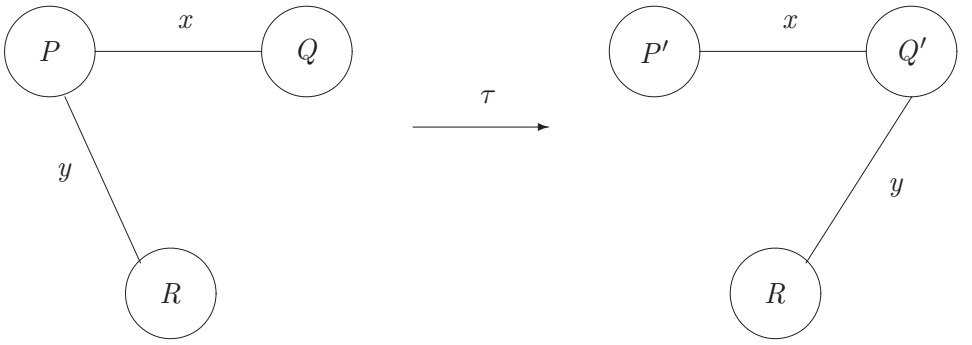


Figura 2.2: Movilidad en el cálculo π.

Los agentes P y Q están conectados mediante el enlace x*, mientras que P y R están conectados mediante el enlace* y*. Inicialmente no existe enlace entre Q y R. Supongamos también que la especificación de los agentes P y Q es la siguiente:*

$$P(x, y) = \bar{x}y.P'(x)$$
$$Q(x) = x(z).Q'(x, z)$$

que indica que el agente P envía el enlace y *a través del enlace* x*, 'olvidando' después dicho nombre de enlace* y*. Q realiza la acción complementaria y consigue de esta forma acceso a R a través de* y*, con lo que el sistema queda como muestra la Figura 2.2 (derecha), pudiendo Q y R comunicarse a partir de este momento por medio del enlace* y *que ambos comparten. Obsérvese cómo mediante una única transición hemos logrado cambiar la topología de comunicación del sistema.* □

2.1.4 Congruencia estructural

La congruencia estructural tiene por objeto identificar expresiones que difieren únicamente desde un punto de vista sintáctico, por ejemplo en cuanto al uso de diferentes nombres ligados o en cuanto a la disposición de los agentes compuestos mediante los operadores de composición paralela o alternativa. La congruencia estructural para el cálculo π aparece definida en varios trabajos, en particular en [Milner, 1991].

Definición 2.2 *La* congruencia estructural*, denotada por $\equiv$, es la mínima relación de congruencia sobre $\mathcal{P}$ tal que*

- $P \equiv Q$ *si ambos difieren únicamente en un cambio de nombres ligados.*
- $(\mathcal{N}/\equiv, +, \mathbf{0})$ *es un monoide simétrico.*
- $(\mathcal{P}/\equiv, \mid, \mathbf{0})$ *es un monoide simétrico.*
- $(x)\mathbf{0} \equiv \mathbf{0}$*, y* $(x)(y)P \equiv (y)(x)P$*.*
- *Si* $x \notin fn(P)$ *entonces* $(x)(P \mid Q) \equiv P \mid (x)Q$*.*

2.1.5 Sustituciones, distinciones y constantes

Las sustituciones, representadas normalmente por la letra σ, se definen de la forma habitual. Así, $P\{y_1/x_1, \ldots, y_n/x_n\}$ indica la sustitución simultánea de todas las ocurrencias libres de los nombres $x_1, \ldots, x_n$ en el proceso P por los nombres $y_1, \ldots, y_n$, respectivamente, con los cambios de nombres ligados que sean necesarios de forma que se evite que alguno de los nombres nuevos y_i quede ligado en P.

Por otro lado, las *distinciones*, definidas como conjuntos de nombres, prohíben la identificación de dichos nombres. Así, diremos que una sustitución σ respeta una distinción D si no liga ningún par de nombres en dicho conjunto, es decir si para todo $x, y \in D$, tal que $x \neq y$ se tiene que $\sigma(x) \neq \sigma(y)$. Como mostraremos más adelante, las distinciones se utilizan en nuestra propuesta para evitar conflictos entre los nombres libres de un agente.

Por último, las constantes se consideran también como nombres en el cálculo π, con la particularidad de que estos nombres no se instancian nunca, considerándose que existe una distinción definida sobre todos ellos. Con objeto de evitar la confusión con cualquier otro nombre en una especificación, representamos las constantes con caracteres versales (CONSTANTE). Por simplicidad, no incluimos las constantes entre los nombres libres de los identificadores que las utilizan, referenciándolas directamente en la especificación de los agentes correspondientes.

2.1.6 Equivalencia y bisimulación

Se han propuesto varias relaciones de equivalencia para este cálculo. Aquí nos referiremos a las de bisimilitud fuerte y débil definidas por Milner, representadas respectivamente por $\sim$ y $\approx$.

Definición 2.3 *Una relación binaria $\mathcal{S}$ es una* simulación (fuerte) *si $P\mathcal{S}Q$ implica que:*

1. *Si* $P \xrightarrow{\tau} P'$*, entonces para algún* $Q', Q \xrightarrow{\tau} Q'$ *y* $P'\mathcal{S}Q'$*.*
2. *Si* $P \xrightarrow{\overline{x}y} P'$*, entonces para algún* $Q', Q \xrightarrow{\overline{x}y} Q'$ *y* $P'\mathcal{S}Q'$*.*

3. Si $P \xrightarrow{x(y)} P'$, $y \notin n(P,Q)$, *entonces para algún* Q', $Q \xrightarrow{x(y)} Q'$ *y* $\forall\, w\ P'\{w/y\}\mathcal{S}Q'\{w/y\}$.

4. Si $P \xrightarrow{\overline{x}(y)} P'$, $y \notin n(P,Q)$, *entonces para algún* Q', $Q \xrightarrow{\overline{x}(y)} Q'$ *y* $P'\mathcal{S}Q'$.

Una relación binaria $\mathcal{S}$ es una *bisimulación* si tanto $\mathcal{S}$ como su inversa son simulaciones. La *bisimilitud de base* (*ground*), representada $\dot{\sim}$, se define como la bisimulación más grande, es decir, aquélla que contiene a cualquier otra bisimulación. Por último, P y Q son *fuertemente bisimilares*, representado $P \sim Q$, si $P\sigma \dot{\sim} Q\sigma$ para cualquier substitución σ.

La distinción entre acciones de salida libres y ligadas, como sucede en el caso de las condiciones segunda y cuarta de la definición anterior, se debe al hecho de que ambas acciones de salida están sometidas a reglas distintas en el sistema de transiciones del cálculo π (recogido en la Figura 2.1). Esto da lugar a la aparición de condiciones ligeramente distintas (con diferencias centradas en los nombres y sustituciones a los que se aplican) para ambos tipos de salidas en la definición de cualquier relación entre agentes en este cálculo.

La necesidad de introducir una relación de bisimilitud *de base* ($\dot{\sim}$) para después generalizarla para cualquier sustitución con el fin de obtener la correspondiente versión completa ($\sim$) viene dada por el hecho de que en el cálculo π la sustitución de nombres libres no preserva la equivalencia, como muestra el siguiente ejemplo.

Ejemplo 2.4 *Consideremos los agentes:*

$x(a).\mathbf{0} \mid \overline{y}(b).\mathbf{0}$
$x(u).\overline{y}(v).\mathbf{0} + \overline{y}(v).x(u).\mathbf{0}$

si consideramos los nombres x *e* y *como distintos se tiene que:*

$$x(a).\mathbf{0} \mid \overline{y}(b).\mathbf{0} \dot{\sim} x(u).\overline{y}(v).\mathbf{0} + \overline{y}(v).x(u).\mathbf{0}$$

Sin embargo, la sustitución de y *por* x *lleva a que :*

$$x(a).\mathbf{0} \mid \overline{x}(b).\mathbf{0} \dot{\not\sim} x(u).\overline{x}(v).\mathbf{0} + \overline{x}(v).x(u).\mathbf{0}$$

puesto que ahora el primero de ellos presenta una transición τ *que el segundo no tiene, al poder sincronizar sus acciones de entrada y salida a través de* x. *Por el contrario, ahora se tiene que:*

$$x(a).\mathbf{0} \mid \overline{x}(b).\mathbf{0} \dot{\sim} x(u).\overline{x}(v).\mathbf{0} + \overline{x}(v).x(u).\mathbf{0} + \tau.\mathbf{0}$$

Por tanto, a la hora de considerar si dos procesos son bisimilares o no, es necesario tener en cuenta las sustituciones de nombres libres, motivo por el cual, la relación de bisimilitud ($\sim$) *se define como la bisimilitud de base* ($\dot{\sim}$) *bajo cualquier sustitución. Así, para el proceso paralelo considerado en este ejemplo se tiene que:*

$$x(a).\mathbf{0} \mid \overline{x}(b).\mathbf{0} \ \sim\ x(u).\overline{x}(v).\mathbf{0} + \overline{x}(v).x(u).\mathbf{0} + [x=y]\tau.\mathbf{0}$$

donde ambos procesos son bisimilares de base independientemente de las sustituciones de nombres libres que consideremos. □

Adicionalmente a la relación de simulación fuerte, existe una versión *débil*, representada como $\approx$, que se obtiene ignorando las acciones silenciosas τ, tal como se muestra a continuación.

Definición 2.5 *Una relación binaria $\mathcal{S}$ es una* simulación débil *si $P\mathcal{S}Q$ implica que:*

1. *Si $P \xrightarrow{\tau} P'$, entonces para algún $Q', Q \Longrightarrow Q'$ y $P'\mathcal{S}Q'$.*
2. *Si $P \xrightarrow{\overline{x}y} P'$, entonces para algún $Q', Q \overset{\overline{x}y}{\Longrightarrow} Q'$ y $P'\mathcal{S}Q'$.*
3. *Si $P \xrightarrow{x(y)} P'$, $y \notin n(P,Q)$, entonces para algún $Q', Q \overset{x(y)}{\Longrightarrow} Q'$ y $\forall\, w\ P'\{w/y\}\mathcal{S}Q'\{w/y\}$.*
4. *Si $P \xrightarrow{\overline{x}(y)} P'$, $y \notin n(P,Q)$, entonces para algún $Q', Q \overset{\overline{x}(y)}{\Longrightarrow} Q'$ y $P'\mathcal{S}Q'$.*

Las definiciones de bisimulación débil, bisimilitud débil de base, y procesos débilmente bisimilares se obtienen del mismo modo que las correspondientes versiones fuertes. Sirva un ejemplo para aclarar las diferencias entre bisimulación fuerte y débil.

Ejemplo 2.6 *Consideremos de nuevo los agentes del ejemplo anterior:*

$$P(x,y) = x(a).\mathbf{0} \mid \overline{y}(b).\mathbf{0}$$
$$Q(x,y) = x(u).\overline{y}(v).\mathbf{0} + \overline{y}(v).x(u).\mathbf{0} + [x=y]\tau.\mathbf{0}$$

Tal como hemos visto, utilizando la Definición 2.3 podemos determinar que $P \sim Q$, puesto que ambos procesos presentan las mismas transiciones: la realización de una acción de entrada a través del enlace x*, la realización de una acción de salida a través del mismo enlace (obsérvese que la diferencia de nombres ligados en ambos agentes no afecta a la simulación), o la realización de una acción interna τ, en el caso de que los nombres* x *e* y *sean el mismo. Además, debido a que la bisimulación fuerte implica a la débil, se tiene también que $P \approx Q$.*

Por otro lado, la Definición 2.5 establece que la diferencia entre bisimulación fuerte y débil está en que esta última no tiene en cuenta la existencia de acciones internas τ (al sustituir las transiciones simples $\longrightarrow$ por transiciones compuestas $\Longrightarrow$). Por tanto, si consideramos ahora el agente:

$$R(x,y) = x(u).\tau.\overline{y}(v).\mathbf{0} + \overline{y}(v).\tau.x(u).\mathbf{0} + [x=y]\tau.\tau.\mathbf{0}$$

aunque ambos procesos no sean fuertemente bisimilares ($Q \nsim R$), por la existencia en el segundo de ellos de transiciones τ que no figuran en el primero, se tiene que $Q \approx R$ (y por supuesto también que $P \approx R$, debido a que tanto $\sim$ como $\approx$ son transitivas).

Sin embargo, no siempre podemos ignorar las transiciones internas a la hora de hablar de bisimulación débil. Así, si consideramos el agente:

$$S(x,y) = \tau.x(u).\overline{y}(v).\mathbf{0} + \tau.\overline{y}(v).x(u).\mathbf{0} + [x=y]\tau.\mathbf{0}$$

se llega a que $Q \not\approx S$, dado que no se cumple la primera condición de la Definición 2.5, puesto que:

$$S(x,y) \xrightarrow{\tau} x(u).\overline{y}(v).\mathbf{0}$$
$$Q(x,y) \overset{\tau}{\not\longrightarrow}$$

y $x(u).\overline{y}(v).\mathbf{0} \not\approx Q(x,y)$, ya que el segundo presenta una acción $\overline{y}(v)$ mientras que el primero no.

Estos dos últimos procesos nos sirven además para mostrar la diferencia entre decisiones globales y locales a la que hacíamos mención a la hora de describir la sintaxis del cálculo en la Sección 2.1.1. En efecto, los agentes Q y S no son débilmente bisimilares debido a que el primero de ellos ofrece un comportamiento alternativo basado en decisiones globales, mientras que el segundo toma de forma local dichas decisiones. Es decir, Q sólo optará por comprometerse con la alternativa encabezada por la acción $\overline{y}(v)$ en combinación con un agente que presente la acción complementaria $x(w)$ —y lo mismo sucede con la alternativa encabezada por $x(u)$—, mientras que S evolucionará realizando una transición τ y se comprometerá con una u otra alternativa —y por tanto con las acciones $x(u)$ o $\overline{y}(v)$—, sin tener en cuenta la presencia o no en su entorno de agentes que ofrezcan acciones complementarias a éstas. Como se ve, el comportamiento de ambos agentes es distinto, en cuanto a la forma en que toman las decisiones, y por este motivo la Definición 2.5 los cataloga como no bisimilares. □

Las definiciones de bisimulación que acabamos de presentar implican la cuantificación universal de las sustituciones, lo que aumenta enormemente el tamaño de las relaciones de bisimulación. Sin embargo, es posible definir un sistema de transiciones más eficiente, tal como se muestra en [Sangiorgi, 1993], lo que permite el desarrollo de algoritmos que comprueben de manera automática las relaciones de bisimilitud [Pistore y Sangiorgi, 1996] y, consiguientemente, de herramientas de análisis.

2.1.7 El cálculo π poliádico

El cálculo π, tal como lo hemos descrito hasta el momento, sólo permite el uso de un único nombre como objeto de una acción de entrada o salida, por lo que estas acciones se denominan *atómicas*, y el cálculo correspondiente se denomina, a su vez, *monádico*. Sin embargo, existe una versión poliádica [Milner, 1991] que permite la comunicación de varios nombres en una única acción. Así, en el cálculo poliádico, las acciones de entrada y salida tienen la forma general:

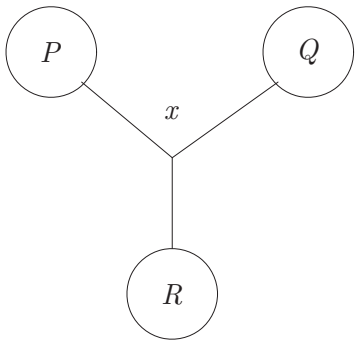
$$\bar{x}\ y_1 \ldots y_n$$
$$x(w_1 \ldots w_n)$$

para indicar, respectivamente, la emisión y recepción de varios nombres a través del enlace x. Por su parte, la emisión y recepción de un *evento* x (es decir, la realización de acciones de salida y entrada a través de un enlace x sin que se use ningún nombre como objeto de dichas acciones), se representa como $\bar{x}$ y x, respectivamente.

Obsérvese que esta transmisión poliádica de varios nombres no corresponde simplemente con una secuencia de acciones en el cálculo monádico, sino que implica una codificación más elaborada. Para ilustrar este punto, consideremos el siguiente ejemplo:

Ejemplo 2.7 (Acciones moleculares) *Supongamos que, tal como muestra la Figura 2.3, tenemos tres agentes P, Q, y R compuestos en paralelo, compartiendo entre todos ellos el enlace* x.

Supongamos ahora que el agente P quiere enviar un par (u,v) *de nombres a* uno *de los agentes Q o R. Un primer intento de codificar estos tres agentes en el cálculo monádico sería el siguiente:*



$$P(x,u,v) \mid Q(x) \mid R(y)$$

Figura 2.3: Tres agentes compartiendo un enlace.

$P(x) = \bar{x}u.\bar{x}v.P'(x,u,v)$
$Q(x) = x(y).x(z).Q'(x,y,z)$
$R(x) = x(y).x(z).R'(x,y,z)$

Sin embargo, al combinar estos tres agentes en paralelo, puede producirse un efecto no deseado. Así, el agente Q puede sincronizar su primera acción de entrada — $x(y)$ — con la primera acción de salida del agente P — $\bar{x}u$ —, y a continuación, R sincronizar su primera acción de entrada — $x(y)$ — con la segunda acción de salida de P — $\bar{x}v$ —, con lo que el sistema evoluciona a:

$P'(x,u,v) \mid$
$x(z).Q'(x,u,z) \mid$
$x(z).R'(x,v,z)$

en el que cada agente Q y R ha recibido uno de los nombres (u,v) *de forma separada. Esta posibilidad puede evitarse utilizando las denominadas* acciones moleculares*. La codificación es ahora la siguiente:*

$P(x) = \bar{x}(w).\bar{w}u.\bar{w}v.P'(x,u,v)$
$Q(x) = x(w).w(y).w(z).Q'(x,y,z)$
$R(x) = x(w).w(y).w(z).R'(x,y,z)$

Como puede apreciarse, la diferencia está en que en lugar de transmitir los nombres u *y* v *directamente, lo que hacemos es que P envíe el nombre de un enlace privado* w *a través del cual se transmitirán posteriormente* u *y* v*. Dado que sólo uno de los procesos receptores Q o R tendrá acceso a este enlace privado, éste será el único proceso que reciba, conjuntamente, los dos nombres en* u *y* v *en cuestión.* □

Este ejemplo muestra que es trivial traducir cualquier codificación poliádica en monádica. En esta memoria, utilizaremos por comodidad codificaciones poliádicas.

2.1.8 Modelización de espacios de tuplas

Como muestra de la capacidad expresiva del cálculo π, hemos realizado la modelización por medio de este formalismo de un conocido ejemplo de estilo arquitectónico, tal como es un espacio de tuplas. Los espacios de tuplas son el mecanismo de comunicación y coordinación entre procesos utilizado por lenguajes de coordinación como Linda [Gelernter, 1985], que hemos descrito en la Sección 1.5.2. La arquitectura de los sistemas de software construidos de acuerdo a este estilo consiste en un conjunto de procesos que se coordinan y comunican escribiendo y leyendo tuplas de un espacio común.

A la hora de representar este estilo arquitectónico por medio de un álgebra de procesos como el cálculo π, podemos establecer las siguientes modelizaciones:

- Modelaremos el espacio de tuplas mediante un agente *Espacio*.
- Modelaremos cada tupla mediante un agente *Tupla*, que será el encargado de almacenar los datos de la tupla y de enviarlos a los procesos que realicen una lectura de la misma.
- Para identificar las tuplas utilizaremos enlaces, de forma que representaremos cada tupla mediante un enlace *tupla*. Este enlace será la forma de acceder al agente *Tupla* correspondiente. Obsérvese que en esta modelización el direccionamiento asociativo de tuplas se reduce a la identidad de los nombres de las tuplas, no existiendo direccionamiento por contenido.
- Las operaciones de escritura y de lectura destructiva y no destructiva de tuplas en el espacio —respectivamente *out*(*tupla*(*datos*)), *in*(*patrón*) y *rd*(*patrón*)— por parte de un proceso se modelan mediante la emisión de un mensaje, desde dicho proceso y con destino el *Espacio*, a través de los enlaces *out*, *in* o *rd* —respectivamente $\overline{out}$ *tupla datos*, $\overline{in}$ *tupla return* y $\overline{rd}$ *tupla return*, siendo *return* el canal de respuesta por el que el proceso recibirá los datos de la tupla.
- La escritura de tuplas activas mediante *eval* se corresponderá con la creación de un proceso en paralelo con el que invoca dicha primitiva.

Una vez definida la modelización del espacio de tuplas y sus primitivas, podemos especificar en el cálculo π el agente *Espacio* como sigue:

$$
\begin{aligned}
&Espacio(in, out, rd) = \\
&\qquad out(tupla\ datos).(\ Tupla(tupla, datos)\ \mid\ Espacio(in, out, rd)\ ) \\
&\quad + rd(tupla\ return).(\ \overline{tupla}\ return.\mathbf{0}\ \mid\ Espacio(in, out, rd)\ ) \\
&\quad + in(tupla\ return).(\ \overline{tupla}\ return\ \textsc{Kill}.\mathbf{0}\ \mid\ Espacio(in, out, rd)\ )
\end{aligned}
$$

$$
\begin{aligned}
&Tupla(tupla, datos) = \\
&\qquad tupla(return).\overline{return}\ datos.Tupla(tupla, datos) \\
&\quad + tupla(return\ kill).[kill = \textsc{Kill}]\ \overline{return}\ datos.\mathbf{0}
\end{aligned}
$$

Como vemos, el agente *Espacio* ofrece tres enlaces —*in*, *out* y *rd*— correspondientes a las tres primitivas de lectura y escritura de tuplas pasivas. A través de estos enlaces, el *Espacio* recibirá instrucciones de los procesos del sistema.

En una operación de escritura, el *Espacio* obtiene a través del enlace *out* el nombre de enlace *tupla* que identifica la tupla a escribir, junto con los *datos* de la misma. A continuación creará un agente *Tupla(tupla,datos)* que representará la tupla escrita.

En una operación de lectura no destructiva —*rd*— el *Espacio* recibe del proceso lector el nombre de enlace que identifica a la tupla, junto con un enlace de retorno. A continuación, el *Espacio* retransmite a la tupla en cuestión el enlace de retorno, para que ésta envíe al proceso lector sus datos a través de dicho enlace.

La operación de lectura destructiva —*in*— es muy similar a la anterior, con la única diferencia que el *Espacio* ordena a la tupla su destrucción a través del parámetro KILL.

Respecto al agente *Tupla*, su comportamiento se reduce a esperar la recepción —a través del enlace *tupla* que lo identifica— de mensajes emitidos por el *Espacio*. Como parámetro de estos mensajes figura el nombre del enlace de retorno que la tupla utilizará para enviar sus datos directamente al proceso lector. En caso de que la operación de lectura sea destructiva (lo que se indica mediante el parámetro KILL), el proceso *Tupla* finaliza.

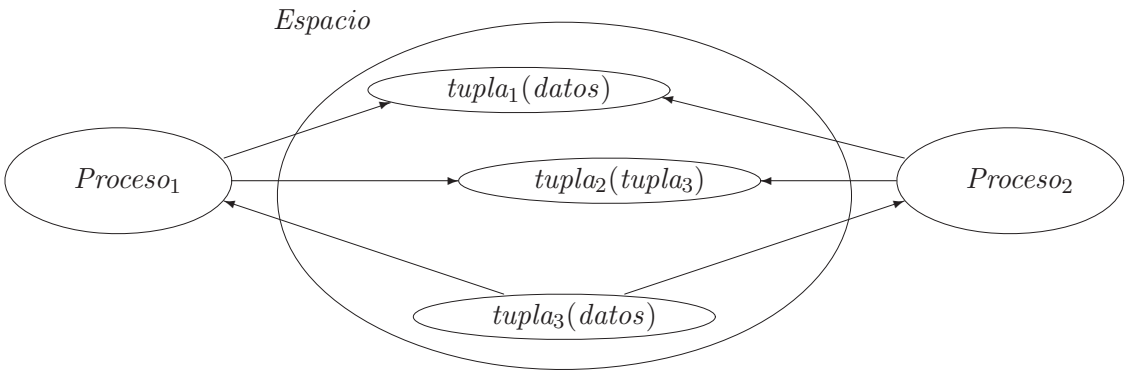


Figura 2.4: Procesos coordinados mediante un espacio de tuplas.

Una vez especificado el espacio de tuplas, podemos mostrar su funcionamiento por medio de un sencillo programa de coordinación, como el que muestra la Figura 2.4 y que está especificado por el agente *Proceso* que presentamos a continuación:

$$
\begin{aligned}
&Proceso(in, out, rd) = (tupla_1, tupla_2) \;\; (\; Proceso_1(in, out, rd, tupla_1, tupla_2) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \mid\; Proceso_2(in, out, rd, tupla_1, tupla_2)\;)
\end{aligned}
$$

$$
\begin{aligned}
Proceso_1(in, out, rd, tupla_1, tupla_2) = \;\; & (datos)\ \overline{out}\ tupla_1\ datos. \\
& (tupla_3)\ \overline{out}\ tupla_2\ tupla_3. \\
& (return)\ \overline{in}\ tupla_3\ return. \\
& return(datos'). \\
& [datos = datos']\ Proceso_1(in, out, rd, tupla_1, tupla_2)
\end{aligned}
$$

$$
\begin{aligned}
Proceso_2(in, out, rd, tupla_1, tupla_2) = \;\; & (ret_1)\ \overline{in}\ tupla_1\ ret_1. \\
& ret_1(datos). \\
& (ret_2)\ \overline{in}\ tupla_2\ ret_2. \\
& ret_2(tupla_3). \\
& \overline{out}\ tupla_3\ datos.Proceso_2(in, out, rd, tupla_1, tupla_2)
\end{aligned}
$$

El agente *Proceso* consiste simplemente en la creación de dos tuplas activas (simulando de este modo el funcionamiento de la primitiva *eval*), representadas por los agentes $Proceso_1$

y $Proceso_2$, respectivamente. Dichos agentes, compuestos en paralelo, comparten los identificadores de tupla $tupla_1$ y $tupla_2$ a través de los cuales se van a coordinar.

El agente $Proceso_1$ se encarga de escribir en el espacio de tuplas una tupla de datos $tupla_1$, seguida de una $tupla_2$ que contiene el identificador de una tercera tupla $tupla_3$. A continuación esperará la presencia en el espacio de una tupla con dicho identificador $tupla_3$ y procederá a su lectura, mediante la primitiva de lectura destructiva *in*. Si los datos leídos en esta tupla coinciden con los originalmente escritos en la $tupla_1$, el agente vuelve a su estado inicial, mientras que si no sucede así, el proceso se bloquea.

Respecto al agente $Proceso_2$, éste se encarga de leer los datos de la $tupla_1$ (si aún no está disponible, esperará a que haya sido escrita por el agente $Proceso_1$). A continuación, obtendrá la $tupla_2$, conteniendo el identificador de la tercera tupla $tupla_3$. Por último, escribe dicha $tupla_3$ conteniendo los datos leídos originalmente de la $tupla_1$ y vuelve a su estado inicial, pasando a repetir todo el proceso.

Para ejecutar este programa de coordinación, bastará con componer en paralelo el espacio de tuplas con el proceso que representa el programa:

$$(in, out, rd)\ (\ Espacio(in, out, rd)\ \mid\ Proceso(in, out, rd)\ )$$

y ejecutar este agente por medio de alguna herramienta de simulación del cálculo π, como puede ser *Mobility Workbench* (MWB) [Victor, 1994]. De esta forma podemos ejecutar sobre el cálculo π programas de coordinación escritos en lenguajes como Linda. En secciones posteriores continuaremos desarrollando este ejemplo.

2.2 Especificación de la arquitectura del software

A medida que las tecnologías de la información se convierten en parte fundamental de las actividades de negocio en todos los sectores industriales, los clientes demandan sistemas más flexibles. En este contexto, el creciente uso de ordenadores personales y el cada vez más fácil y generalizado acceso a redes de comunicación globales y locales proporcionan una infraestructura excelente para el desarrollo de sistemas abiertos y distribuidos. Sin embargo, los problemas específicos que plantean este tipo de sistemas, dinámicos y de gran tamaño, suponen un reto para la Ingeniería del Software, cuyos métodos y herramientas tradicionales se topan con dificultades para abordar los nuevos requisitos.

Dentro de este contexto general, nuestro enfoque se basa en el uso del cálculo π para la especificación de la arquitectura del software. Además de la posibilidad de realizar análisis formal que está presente en las álgebras de proceso, la expresión directa de la movilidad en el cálculo π permite la representación de arquitecturas que presenten una topología de comunicación cambiante, lo que sólo puede hacerse con muchas limitaciones utilizando otros formalismos como CSP o CCS, tal como se muestra en [Allen et al., 1998]. De hecho, el tipo de reconfiguración dinámica presente en los sistemas abiertos y distribuidos se expresa de modo muy natural en el cálculo π por medio de la transmisión de nombres de enlaces entre agentes, del mismo modo que los componentes de software de un sistema dinámico y distribuido intercambian referencias a componentes, *sockets* o direcciones URL.

2.2.1 Nociones intuitivas

Para realizar la representación de un sistema de software en el cálculo π, podemos pensar en especificar cada uno de los componentes de este sistema mediante un agente, y a continuación, componer en paralelo estas especificaciones. A este respecto, es preciso señalar que aunque representemos la composición de componentes de software mediante el operador paralelo (|), nuestra propuesta no se dirige únicamente a la especificación y validación de arquitecturas formadas por un conjunto de componentes actuando de forma concurrente. Por el contrario, la composición paralela es la representación natural, dentro de las álgebras de proceso, de cualquier forma de combinación de componentes de software. Por tanto, incluso los sistemas secuenciales se representan por medio de la composición paralela de sus componentes.

Por otro lado, las conexiones entre componentes pueden realizarse mediante la identificación de nombres de enlaces en la especificación de los correspondientes componentes, de tal forma que se permita la comunicación entre ellos. Esta primera aproximación a la especificación de la arquitectura del software utilizando el cálculo π se muestra en el siguiente ejemplo:

Ejemplo 2.8 (Concentrador) *Pensemos en un componente* Concentrador *que combina las entradas recibidas por dos canales, e_1 y e_2, reenviándolas por un único canal de salida s:*

$$\begin{aligned} Concentrador(e_1, e_2, s) = {} & e_1(data).\bar{s}data.Concentrador(e_1, e_2, s) \\ + {} & e_2(data).\bar{s}data.Concentrador(e_1, e_2, s) \end{aligned}$$

y supongamos ahora dos componentes adicionales, un Emisor *y un* Receptor*:*

$$Emisor(s) = \bar{s}(data).Emisor(s)$$
$$Receptor(e) = e(data).Receptor(e)$$

El sistema formado por la combinación de dos emisores, un concentrador y un receptor vendría especificado por el siguiente proceso:

$$(e_1, e_2, s)(\ Emisor(e_1)\ \mid\ Emisor(e_2)\ \mid\ Concentrador(e_1, e_2, s)\ \mid\ Receptor(s)\ )$$

donde el operador de composición paralela es usado para modelar la combinación de los componentes unos con otros, mientras que la identificación de nombres de enlaces en los parámetros de estos componentes determina las conexiones que se establecen entre ellos. Analizando este uso de nombres de enlaces compartidos, podemos determinar la estructura del sistema, que aparece representada en la Figura 2.5. □

No obstante, y a pesar de ser intuitivo, este enfoque presenta dos claras limitaciones. En primer lugar, la arquitectura del sistema, que se deriva de las relaciones que cada componente mantiene con el resto, no se muestra de forma explícita, sino que queda oculta en la compartición de nombres de parámetros entre los distintos agentes. En segundo lugar, y no menos importante, una especificación de este tipo imposibilitaría el análisis de sistemas complejos, debido a la explosión del número de estados, ya que implica la especificación completa de los componentes.

Para resolver estas limitaciones, en lugar de realizar una especificación completa de los componentes, utilizaremos especificaciones parciales de interfaz, o *roles*, para la descripción del comportamiento de cada componente, de forma que este comportamiento vendrá indicado por

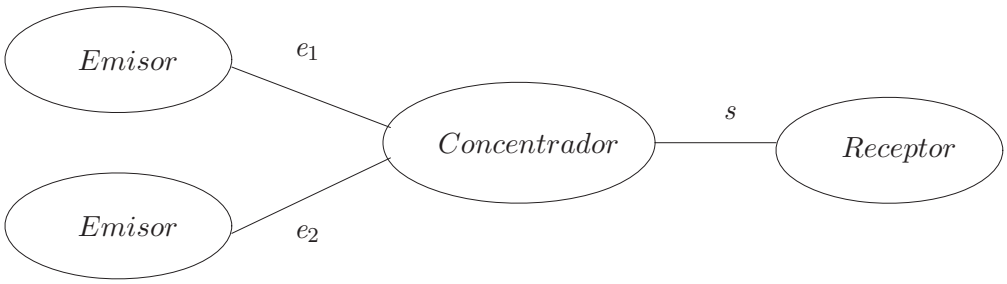


Figura 2.5: Sistema con dos emisores, un concentrador y un receptor.

los papeles o roles que dicho componente juegue en el sistema. Por otro lado, representaremos de forma explícita la arquitectura del sistema como un conjunto de *conexiones* entre roles de los componentes del mismo. A continuación, analizaremos localmente cada una de las conexiones, es decir, determinaremos la conformidad o disconformidad entre los componentes del sistema mediante un análisis de la compatibilidad de los roles que los representan. De esta forma se reduce en gran medida la complejidad del análisis.

Como veremos, este tipo de análisis local no permite deducir determinadas propiedades globales, como por ejemplo que el sistema en su conjunto esté libre de bloqueos. Sin embargo, no es éste nuestro objetivo. Tal como demostraremos, el análisis local de la compatibilidad garantiza la ausencia de bloqueos en la unión de un componente con los roles especificados en una determinada arquitectura, asegurando de este modo que el componente 'encaja' en la arquitectura y que no hay desajustes (*mismatch*) entre las interfaces de los distintos componentes. Opinamos que esto es suficiente para considerar el que el sistema es *composicional*. Aquellos roles que sean compatibles serán capaces de interactuar correctamente, indicando la compatibilidad de los respectivos componentes. Por otro lado, una incompatibilidad detectada a la hora de analizar una conexión entre roles, indica una disconformidad en el comportamiento de los componentes correspondientes, que por lo general llevará a un fallo o caída del sistema de software.

2.2.2 Formalización del concepto de rol

En la sección precedente, hemos presentado de forma intuitiva los conceptos arquitectónicos que van a ser claves en nuestro trabajo. A continuación procederemos a formalizar dichos conceptos en el contexto del cálculo π. La definición de rol que figura más abajo se basa en la noción de abstracción en álgebras de procesos, establecida por primera vez en [Boudol, 1987]. Nuestra definición es una adaptación para el cálculo π basada en un operador de *ocultación* ($/_{\tilde{x}}$) similar al definido en [Milner, 1989] para CCS, pero adaptado a las características de movilidad presentes en el cálculo π:

Definición 2.9 (Ocultación) *Sea P un agente, y sea $\tilde{x} \subseteq fn(P)$. La* ocultación *en P de los nombres $\tilde{x}$, representada como $P/_{\tilde{x}}$, se define como:*

$$P/_{\tilde{x}} = (\tilde{x})(P \mid \prod_{x \in \tilde{x}} Siempre(x))$$

donde $Siempre(x) = x(y).(Siempre(x) \mid Siempre(y)) + (y)(\bar{x}y.(Siempre(x) \mid Siempre(y)))$

El agente *Siempre*(x) es el que se encarga de realizar la ocultación de cada nombre $x \in \tilde{x}$ en P. Este proceso estará siempre dispuesto a recibir o enviar mensajes a través del enlace x, con lo que satisfará cualquier necesidad de comunicación de P a través de dicho enlace (impidiendo por tanto que P se bloquee), así como a través de cualquier otro recibido o emitido a través de él. De esta forma, las acciones que A presenta a través del enlace x se muestran en el agente $P/_x$ que oculta dicho nombre, como transiciones silenciosas τ.

Llegados a este punto, podemos definir los roles como sigue:

Definición 2.10 (Rol) *Un agente P es un* rol *de un componente Comp si*

$$fn(P) \subseteq fn(Comp) \quad y \quad P \approx Comp/_{(fn(Comp)-fn(P))}$$

Según las definiciones anteriores, los nombres libres de un rol son un subconjunto de los nombres libres del componente correspondiente. Así por ejemplo, podemos dividir la interfaz del componente *Concentrador* del Ejemplo 2.8, en tres roles, uno por cada uno de sus nombres libres. Cada uno de estos roles se limita a describir la interfaz del componente según uno de los tres enlaces que figuran como parámetros del *Concentrador*, haciendo abstracción de las acciones que se realizan a través de los otros enlaces. Por tanto, los roles se obtienen de los componentes *ocultando* los nombres que no son relevantes para la interfaz parcial representada por el rol. De esta forma, cualquier acción del componente que utilice nombres ocultos, aparecerá en el rol como una transición silenciosa, y un rol sólo contiene un subconjunto de las transiciones no silenciosas del componente correspondiente. Nótese que la Definición 2.10 no identifica un único agente, sino que está realizada módulo bisimulación débil. Por este motivo, dado un componente y un subconjunto de sus nombres libres, puede obtenerse todo un conjunto de roles, formando una clase de equivalencia. Sin embargo, no todos estos agentes serán representaciones significativas o incluso correctas de la interfaz del componente. Abordaremos el tema de la corrección de los roles de un componente en la Definición 2.16.

Ejemplo 2.11 *Según lo anteriormente expuesto, la especificación del rol correspondiente al enlace e_1, obtenida a partir del* Concentrador*, sería la siguiente:*

$$Rol(e_1) = e_1(data).\tau.Rol(e_1) + \tau.\tau.Rol(e_1)$$

Esta especificación puede ser simplificada haciendo uso de la relación de bisimilitud débil, obteniéndose el rol:

$$Rol'(e_1) = e_1(data).Rol'(e_1)$$

que representa el comportamiento del Concentrador *según es observado desde el enlace e_1. Si consideramos ahora el enlace e_2 llegaríamos a una especificación idéntica para el rol correspondiente a dicho enlace. Nótese que la especificación de ambos roles coincide con la del agente* Receptor *del Ejemplo 2.8. Por tanto, podemos decir que, en su relación con los componentes* Emisor*, el* Concentrador *juega el papel de* Receptor*. De forma similar, al derivar el rol correspondiente al enlace s, llegaríamos a la especificación del agente* Emisor*, papel que desempeña el* Concentrador *en relación con el componente* Receptor *situado a su salida. De aquí en adelante, consideraremos que el interfaz del componente* Concentrador *consta de dos instancias del rol* Receptor *y una instancia del rol* Emisor*.*

Podemos representar gráficamente al componente Concentrador *junto a sus tres roles, tal como se muestra en la Figura 2.6. La notación utilizada representa a los componentes mediante óvalos, a los roles mediante círculos, y la relación rol-componente se indica mediante trazos discontinuos.*

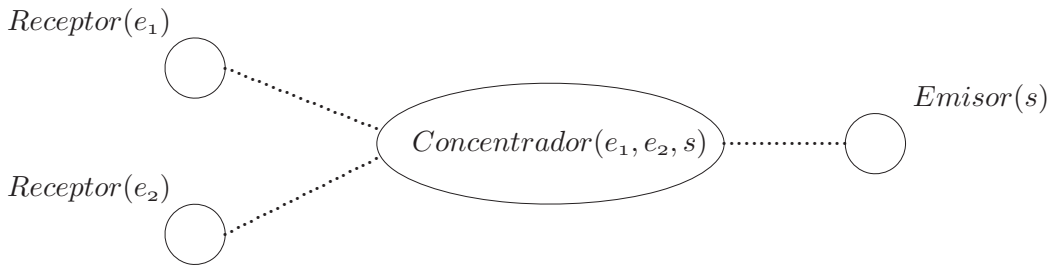


Figura 2.6: Componente *Concentrador* con sus roles.

Este mismo ejemplo sirve para mostrar cómo los roles representan una visión parcial *de la interfaz de sus respectivos componentes. Así, si consideramos la interfaz que proporciona a su entorno el componente* Concentrador *en su conjunto, esta interfaz incluye secuencias de acciones como la siguiente:*

$$Concentrador(e_1, e_2, s) \ : \ e_1(d).\overline{s}d.e_2(d).\overline{s}d.e_2(d).\overline{s}d.e_1(d).\overline{s}d \ \ldots$$

mientras que si consideremos sus tres roles por separado, la interfaz ofrecida por estos incluye trazas de la forma:

$$\begin{array}{ll} Receptor(e_1) : & e_1(d).e_1(d) \ \ldots \\ Receptor(e_2) : & e_2(d).e_2(d) \ \ldots \\ Emisor(s) \ : & \overline{s}d.\overline{s}d.\overline{s}d.\overline{s}d \ \ldots \end{array}$$

es decir, cada uno de estos roles ofrece una visión parcial del comportamiento del componente. □

En el ejemplo anterior hemos visto cómo las acciones del componente que utilizan nombres libres ocultados por el rol aparecen en este último como transiciones silenciosas, transiciones que, por lo general, pueden ser eliminadas en virtud de la relación de bisimulación débil. No obstante, si alguna de estas acciones ocultadas por un rol encabezaba una alternativa del componente combinada con otras mediante el operador suma (ofreciendo por tanto el componente a su entorno una *decisión global* en función de dicha acción), esto daría lugar a la aparición en el rol de una *decisión local*, debido a que, desde el punto de vista de otro componente conectado a este rol, no es posible determinar si dicha alternativa será o no elegida. El siguiente ejemplo es ilustrativo de la situación que acabamos de describir.

Ejemplo 2.12 (Traductor) *Supongamos un* Traductor*, un componente que copia los datos recibidos por sus enlaces de entrada e_1 y e_2 a los respectivos enlaces de salida s_1 y s_2, de forma que realiza una suerte de traducción entre los nombres de enlace de entrada y salida.*

$$\begin{aligned} Traductor(e_1, e_2, s_1, s_2) = \; & e_1(x).\overline{s_1}x.Traductor(e_1, e_2, s_1, s_2) \\ + \; & e_2(y).\overline{s_2}y.Traductor(e_1, e_2, s_1, s_2) \end{aligned}$$

De forma similar a como hicimos en el ejemplo anterior, dividiremos la interfaz de este componente en dos roles, que denominaremos respectivamente Entrada *y* Salida*, con nombres libres* e_1, e_2 *y* s_1, s_2 *respectivamente. A partir de la Definición 2.10, se llegaría a la siguiente especificación del rol* Salida*:*

$$Salida(s_1, s_2) = \tau.\overline{s_1}(x).Salida(s_1, s_2) + \tau.\overline{s_2}(y).Salida(s_1, s_2)$$

en la que se aprecia la presencia de una decisión local. Esto es debido a que si observamos el Traductor *desde el punto de vista del componente situado a su salida (es decir, si consideramos su rol* Salida*), no podemos predecir cuál de las dos acciones —*$\overline{s_1}(x)$ *o* $\overline{s_2}y$*— va a realizar. La acción finalmente elegida dependerá de cuál haya sido la acción de entrada inmediatamente anterior, que forma parte de un rol diferente, y por tanto estará oculta para el rol* Salida*. Estas acciones de entrada del componente quedan especificadas en el rol por medio de acciones* τ*, configurando de esta forma la ya citada decisión interna.*

Por último, merece la pena llamar la atención sobre el hecho de que las acciones de salida a través de los enlaces s_1 *y* s_2 *aparecen como acciones libres en la especificación del componente, mientras que dichas acciones figuran como salidas ligadas en el rol* Salida*. Esta transformación de acciones de salida libres en salidas ligadas es sutil, pero ocurre con cierta frecuencia, por lo que merece un comentario. Obsérvese que los nombres* x *e* y *usados como objetos en las acciones de salida del* Traductor*, se obtienen como resultado de las acciones de entrada realizadas previamente a través de los enlaces* e_1 *y* e_2*. Por tanto, dichas acciones de salida son libres en el componente.*

Sin embargo, en el rol Salida*, los enlaces* e_1 *y* e_2 *están ocultos, y las acciones de entrada a través de dichos enlaces aparecen simplemente como acciones internas* τ*. Por tanto, los nombres usados en las acciones de salida del rol deben ser considerados como nuevos, con lo que dichas acciones son ahora ligadas. Esta transformación, que es acorde con la Definición 2.10, no sólo tiene sentido atendiendo a aspectos técnicos del cálculo* π*, sino también de forma intuitiva: desde el punto de vista de un observador conectado al rol* Salida *del* Traductor*, los nombres* x *e* y *recibidos de este componente son nuevos, y no pueden ser rastreados hasta su origen en las acciones de entrada ocultas a través de los enlaces* e_1 *y* e_2*.* □

De forma similar, las acciones de un componente que estén restringidas por guardas sobre nombres ocultos, también aparecerán como decisiones locales en el rol correspondiente. Una vez más, el motivo es que, desde el punto de vista del entorno del componente, el compromiso con una de esas transiciones se realiza por medio de una decisión local. Esta transformación de guardas en transiciones silenciosas se realiza en dos pasos. Tal como se indica en [Milner et al., 1992], las guardas sobre tipos de datos no son necesarias en el cálculo π, y pueden ser sustituidas por acciones a través de los nombres involucrados. Entonces, debido a que estas acciones utilizan nombres ocultos, se abstraen en el rol, donde quedan representadas mediante transiciones silenciosas. De nuevo podemos ilustrar este punto con un ejemplo:

Ejemplo 2.13 (Observador) *Supongamos un* Observador*, un componente que recibe bolas de colores de un componente* GeneradorDeBolas*, y que emite eventos (acciones de salida sin datos) a través de los enlaces* roja *o* negra*, dependiendo del color de la bola. Estos dos componentes pueden especificarse, utilizando guardas, de la siguiente manera:*

$$Observador(bola, roja, negra) = bola(color).$$
$$(\ [color = \text{ROJA}]\ \overline{roja}.Observador(bola, roja, negra)$$
$$+\ [color = \text{NEGRA}]\ \overline{negra}.Observador(bola, roja, negra)\ )$$

$$GeneradorDeBolas(bola) = \tau.\overline{bola}\ \text{ROJA}.GeneradorDeBolas(bola)$$
$$+\ \tau.\overline{bola}\ \text{NEGRA}.GeneradorDeBolas(bola)$$

pero resulta sencillo obtener una codificación equivalente de estos dos componentes sin hacer uso de guardas:

$$Observador(bola, roja, negra) = bola(color).color(colorRojo\ colorNegro).$$
$$(\ colorRojo.\overline{roja}.Observador(bola, roja, negra)$$
$$+\ colorNegro.\overline{negra}.Observador(bola, roja, negra)\ )$$

$$GeneradorDeBolas(bola) = \overline{bola}(nueva).\overline{nueva}(roja\ negra).$$
$$(\ \tau.roja.GeneradorDeBolas(bola) + \tau.negra.GeneradorDeBolas(bola)\ )$$

si ahora consideramos el rol Salida(roja, negra) *que especifica el comportamiento del* Observador *según es visto desde sus enlaces de salida* roja *y* negra*, utilizando la Definición 2.10 obtendremos:*

$$Salida(roja, negra) = \tau.\overline{roja}.Salida(roja, negra) + \tau.\overline{negra}.Salida(roja, negra)$$

donde hemos hecho abstracción de las guardas que figuraban en el Observador *original, guardas que quedan ahora representadas mediante transiciones silenciosas.* □

2.2.3 Corrección de los roles de un componente

Según hemos visto, los roles se especifican mediante agentes en el cálculo π, por lo que para determinar el éxito de la composición de una serie componentes, habremos de ser capaces de diferenciar entre bloqueo y terminación con éxito de un agente, es decir, entre agentes que *son* la inacción, como $\mathbf{0} \mid \mathbf{0}$, y aquéllos que *se comportan* como la inacción, como $(a, b)(a(w).\mathbf{0} \mid \overline{b}y.\mathbf{0})$.

Desgraciadamente no existe en el cálculo π más que una forma de indicar la terminación de un proceso, y esta forma es mediante el uso de la inacción $\mathbf{0}$. Cualquier proceso que esté imposibilitado de realizar más acciones (ya sea porque esté bloqueado o porque haya terminado) es equivalente a la inacción. El motivo es que en este álgebra de procesos los procesos terminados y los bloqueados son indistinguibles, desde el punto de vista de un observador. Esto es debido a que el cálculo π no presenta un operador de composición secuencial entre procesos (que sí existe en otras álgebras, como ACP o CSP). Es precisamente este operador el que hace necesario distinguir explícitamente entre terminación y bloqueo. En efecto, consideremos los procesos en ACP $nil \cdot P$ y $\delta \cdot Q$ (donde '$\cdot$' representa el operador de composición secuencial, *nil* se utiliza para representar un proceso terminado con éxito, y δ indica un proceso bloqueado). Resulta evidente que $nil \cdot P$ puede realizar cualquier acción que P sea capaz de realizar, mientras que es natural asumir que $\delta \cdot Q$ está bloqueado y no puede realizar ninguna acción. Por tanto, en presencia de composición secuencial hay una diferencia observable entre la terminación con éxito y el fracaso que, diferencia que, como ya hemos dicho, no es en general relevante en el cálculo π.

Las nociones de terminación, bloqueo y divergencia en álgebras de procesos han sido abordadas en diversos trabajos, y de forma específica en [Aceto y Hennessy, 1992]. Siguiendo un enfoque similar al de estos autores, presentamos aquí una definición de éxito y fracaso en el contexto del cálculo π en la que la congruencia estructural juega un papel importante para distinguir entre bloqueo y terminación con éxito. Consideraremos que un agente P tiene éxito si alcanza la inacción después de una secuencia (puede que vacía o infinita) de transiciones silenciosas τ. La definición siguiente formaliza esta idea:

Definición 2.14 (Éxito y fracaso) *Un agente P es un* fracaso *si existe P' tal que $P \Longrightarrow P'$, $P' \stackrel{\tau}{\not\longrightarrow}$, y $P' \not\equiv \mathbf{0}$.*

Por el contrario, decimos que un agente tiene éxito*, si no es un fracaso.*

Por tanto, consideraremos que tienen éxito aquellos agentes que siempre progresen hacia la inacción sin precisar de interacción con su entorno, y como fracasos aquéllos que, en las mismas condiciones, se bloquearían.

En la Definición 2.10, los roles se definen módulo la relación de bisimulación débil. Por tanto, dado un componente *Comp* y un conjunto de nombres $\mathcal{M}$ tal que $\mathcal{M} \subseteq fn(Comp)$, podemos derivar varios roles bisimilares débilmente. Sin embargo, no todos ellos representan al componente de forma correcta, tal como muestra el siguiente ejemplo.

Ejemplo 2.15 *Supongamos un componente:*

$$Comp(a,b) = a(x).Comp(a,b) + b(y).\mathbf{0}$$

y dos agentes:

$$P_1(a) = a(x).P_1(a) + \tau.\mathbf{0}$$
$$P_2(b) = b(y).(c)c.\mathbf{0}$$

De acuerdo a la Definición 2.10, tanto $P_1(a)$ como $P_2(b)$ son roles de $Comp(a,b)$, ocultando respectivamente los nombres b y a. Sin embargo, mientras que $P_1(a)$ representa correctamente el comportamiento de Comp(a,b) respecto al nombre a, $P_2(b)$ no hace lo mismo respecto a b, ya que la acción $b(z)$ lleva a Comp(a,b) al éxito y a $P_2(b)$ al fracaso. □

Como acabamos de ver, la Definición 2.10 es tan sólo una condición necesaria para que un agente constituya un rol de un componente. El objetivo de la definición que sigue es determinar la corrección de los roles de un componente, estableciendo una condición suficiente para ello.

Definición 2.16 (Corrección) *Sea Comp un componente y $\mathcal{P} = \{P_1, \ldots, P_n\}$ un conjunto de roles de dicho componente.* $\mathcal{P}$ especifica correctamente *a Comp si y sólo si*

1. $fn(P_i) \cap fn(P_j) = \varnothing \quad \forall\, i \neq j$
2. $Comp \Longrightarrow \mathbf{0}$ *si y sólo si* $\forall\, i\ P_i \Longrightarrow \mathbf{0}$
3. *Si* $\exists\, \alpha\ (\alpha \neq \tau)$ *tal que* $Comp \stackrel{\alpha}{\Longrightarrow} Comp'$*, entonces* $\exists\, i, P_i'$ *tal que* $P_i \stackrel{\alpha}{\Longrightarrow} P_i'$*, y donde* $\mathcal{P}' = \{P_1, \ldots, P_i', \ldots, P_n\}$ *especifica correctamente a* $Comp'$*.*

La Definición 2.16 indica que la interfaz del componente *Comp* debe estar completamente especificada por los roles en $\mathcal{P}$, y los conjuntos de nombres libres de los roles han de ser disjuntos. Esto último asegura que la especificación del componente se realice de manera modular, de forma que los diferentes roles no puedan sincronizar unos con otros. Así, si

$$P_1 \mid \cdots \mid P_i \mid \cdots \mid P_n \xrightarrow{\tau} P$$

entonces algún P_i tendrá una τ transición a P_i', donde

$$P = P_1 \mid \cdots \mid P_i' \mid \cdots \mid P_n$$

Análogamente, si

$$P_1 \mid \cdots \mid P_i \mid \cdots \mid P_n \xrightarrow{\alpha} P \ (\alpha \neq \tau)$$

exactamente *un* P_i tendrá una transición α a P_i', donde

$$P = P_1 \mid \cdots \mid P_i' \mid \cdots \mid P_n$$

Las Definiciones 2.10 y 2.16 hacen posible la derivación de roles a partir de la especificación de los componentes correspondientes.

Según la definición de corrección que acabamos de presentar, la interfaz de un componente quedará especificada mediante un conjunto de roles, que describen el comportamiento del componente tal como es observado desde su entorno. Estos roles habrán de ser *disjuntos*, en el sentido de que cada uno de ellos hará referencia a la interacción del componente con otro componente del sistema, haciendo abstracción de comportamientos que no sean relevantes para el rol en cuestión.

No debemos olvidar que el objetivo de especificar roles no es otro que el de describir la interfaz del componente, es decir, únicamente el comportamiento observable desde su entorno, y además realizar está descripción de forma modular: cada uno de los roles describe el comportamiento relativo a un conjunto de enlaces, ignorando el resto, por lo que ofrece únicamente una visión parcial, la que interesa a aquel componente conectado a éste precisamente a través de dichos enlaces. Como ya se ha indicado anteriormente, la razón para prescindir de las computaciones internas y de las relaciones entre los distintos roles de la interfaz no es otra que simplificar el análisis de la arquitectura del software.

Por este motivo, no podemos considerar un conjunto de roles que especifique a un componente como una especificación *total* del mismo, es decir, de sus computaciones internas o de las interdependencias entre las acciones de comunicación por medio de enlaces que figuran en roles distintos, sino sólo de su interfaz.

El ejemplo que figura a continuación muestra estas diferencias de puntos de vista entre el comportamiento que define la especificación de un componente y el indicado por sus roles.

Ejemplo 2.17 (Variable) *Consideremos una variable del tipo de las usadas en un lenguaje de programación imperativo. Sobre ella se pueden realizar operaciones de lectura y escritura, habiendo de comenzar por una escritura que fije su valor inicial. La especificación de este componente es la siguiente:*

$$Variable(leer, escribir) = \\ escribir(valorinicial).VariableInicializada(leer, escribir, valorinicial)$$

$$VariableInicializada(leer, escribir, valor) = \\ \overline{leer}(valor).VariableInicializada(leer, escribir, valor) \\ + escribir(nuevovalor).VariableInicializada(leer, escribir, nuevovalor)$$

Este sencillo componente juega dos roles diferentes en un programa: de almacenamiento y de recuperación de datos. De este modo, la interfaz de la Variable *se divide en dos roles* —Almacenamiento *y* Recuperación—*, en los cuales las acciones referidas a nombres ocultos estarían representadas por transiciones τ. Debido a que estas acciones internas no determinan la existencia de decisiones locales, pueden ser omitidas en dichos roles, obteniéndose versiones débilmente equivalentes de ellos como las que figuran a continuación, que especifican correctamente la* Variable*.*

$$Almacenamiento(escribir) = escribir(nuevovalor).Almacenamiento(escribir)$$

$$Recuperación(leer) = \overline{leer}(valor).Recuperación(leer)$$

Como se puede apreciar, cada uno de estos roles da una visión parcial e incompleta de la interfaz del componente, y ni siquiera considerados en su conjunto describen los roles completamente al componente. En efecto, la división del comportamiento de la Variable *en dos roles, referidos respectivamente a las operaciones de escritura y lectura, hace que ya no quede de manifiesto el requisito de que para poder realizar una operación de lectura, la* Variable *tiene que haber sido inicializada primero. En este sentido, debemos considerar el ofrecimiento de acciones $\overline{leer}(valor)$ que se realiza en el rol* Recuperación *con el significado de que* en algún momento *el componente ofrecerá la posibilidad de realizar dicha acción de lectura (una vez que haya sido inicializado mediante una operación de escritura, momento que no puede ser precisado por quien observa la* Variable *únicamente a través del enlace* leer*).*

Siguiendo con nuestro ejemplo, supongamos ahora que queremos utilizar nuestro componente Variable *en un sistema Lector/Escritor, en donde actuará como variable compartida para el intercambio de datos entre dos procesos, uno que actuará de* Lector *y otro que lo hará de* Escritor*, de acuerdo, respectivamente, a los roles:*

$$Lector(leer) = leer(valor).Lector(leer)$$

$$Escritor(escribir) = \overline{escribir}(nuevovalor).Escritor(escribir)$$

En un sistema de este tipo, los roles Almacenamiento *y* Recuperación *que hemos definido representarán a la* Variable *en sus conexiones con los roles* Escritor *y* Lector *que representen a los programas. Estas conexiones entre roles serán suficientes para describir y analizar la arquitectura del sistema de Lector/Escritor, sin necesidad de razonar sobre la especificación completa de la variable, ni tampoco de los procesos representados por los roles* Lector *y* Escritor*.* □

2.2.4 Formalización de los conceptos de conexión y arquitectura

La composición de varios componentes en una cierta arquitectura será representada mediante una conexión entre roles de dichos componentes. Con objeto de evitar la sincronización entre

conexiones distintas, los nombres libres de estos roles deben estar convenientemente restringidos. Por tanto, las conexiones se definen de la forma siguiente:

Definición 2.18 (Conexión) *Sean $P_1, \ldots, P_n$ una serie de roles. Su* conexión *se define como el conjunto:*

$$\{P_1, \ldots, P_n\}$$

(conjunto que, por lo general, escribiremos de forma abreviada como $\{P_i\}_i$). Asociado a una conexión $\{P_i\}_i$, existe un agente que la representa, definido como:

$$(\bigcup_i fn(P_i))(\prod_i P_i)$$

Por último, podemos definir una *arquitectura*, formada por la composición de varios componentes, como un conjunto de conexiones entre roles de dichos componentes.

Definición 2.19 (Arquitectura) *Considérese un sistema de software formado por varios componentes $\{Comp_j\}_{j=1}^{n}$. Sean $\mathcal{R}_j = \{R_{j_i}\}_{i=1}^{n_j}$ los roles que especifican correctamente cada $Comp_j$*
$(j = 1 \ldots n)$. Entonces, una arquitectura *del sistema se define como una partición disjunta Ψ de $\mathcal{R}oles = \cup_{j=1}^{n}\mathcal{R}_j$, representando las conexiones entre roles que forman el sistema a partir de sus componentes $\{Comp_j\}_j$. Es decir,*

$$\Psi = \{\mathcal{R}oles_1, \ldots, \mathcal{R}oles_m\} \text{ tal que } \mathcal{R}oles = \dot{\bigcup}_{k=1}^{m}\mathcal{R}oles_k$$

Asociado a una arquitectura Ψ como la descrita, existe un agente que la representa, definido como:

$$\prod_{k=1}^{m} Roles_k$$

donde $Roles_k$ es el agente que representa a la conexión $\mathcal{R}oles_k$, $(k = 1 \ldots m)$.

Con objeto de simplificar algunos de los resultados de las secciones siguientes, consideraremos primero arquitecturas *binarias*, en las que cada conexión $\mathcal{R}oles_k$ conecte solamente un par de roles. Posteriormente, estos resultados se extenderán a arquitecturas más generales.

Ejemplo 2.20 *Volviendo al sistema del Ejemplo 2.8, formado por dos componentes* Emisor*, un* Concentrador *y un* Receptor*, la conexión entre el rol* Emisor *del* Concentrador *y el componente* Receptor *(representado por sí mismo), vendría dada por el conjunto:*

$$\{\ Emisor(s), Receptor(s)\ \}$$

que lleva asociado el agente:

$$(s)(Emisor(s)\ \mid\ Receptor(s))$$

Del mismo modo, y según la Definición 2.19 que acabamos de ofrecer, la arquitectura de este sistema se describiría como:

$\{ \{ Emisor(e_1), Receptor(e_1) \},$
$\{ Emisor(e_2), Receptor(e_2) \},$
$\{ Emisor(s), Receptor(s) \} \}$

conjunto que lleva asociado el agente:

$(e_1)(Emisor(e_1) \mid Receptor(e_1)) \mid$
$(e_2)(Emisor(e_2) \mid Receptor(e_2)) \mid$
$(s)(Emisor(s) \mid Receptor(s))$

Podemos representar gráficamente dicha arquitectura tal como se muestra en la Figura 2.7, en la que los componentes aparecen representados mediante óvalos, los roles mediante círculos, las relaciones entre un componente y sus roles mediante trazos discontinuos y las conexiones entre roles, mediante trazos continuos. □

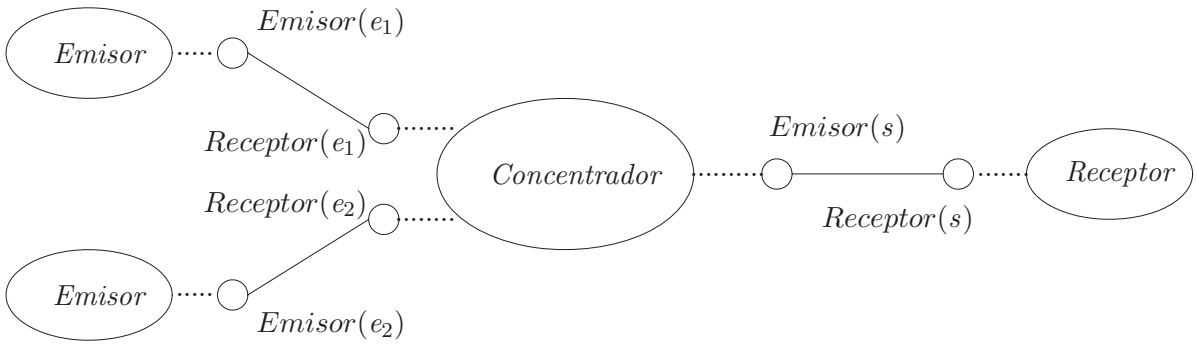


Figura 2.7: Representación gráfica de la arquitectura del sistema del Ejemplo 2.8.

Haciendo un cierto abuso del lenguaje, a partir de este momento utilizaremos indistintamente el término *conexión* para referirnos tanto al conjunto que forman una serie de roles interconectados como al agente asociado a dicha conexión y que la representa, y utilizaremos el término *arquitectura* para referirnos tanto a un conjunto de conexiones como al agente que representa dicha arquitectura, siendo el contexto donde dichos términos sean utilizados el que determine si nos referimos a un conjunto o a un agente.

2.3 Compatibilidad de comportamiento

La progresiva implantación de nuevas tecnologías para el desarrollo de software, y en concreto la Ingeniería del Software Basada en Componentes, está haciendo cambiar la forma en la que se construye actualmente el software. Las organizaciones de software están pasando gradualmente de realizar *desarrollo* de aplicaciones a, cada vez más, basarse en el *ensamblado* de componentes software preexistentes. Esto hace que los esfuerzos de desarrollo se vayan centrando de manera progresiva en la localización de componentes, identificación de sus capacidades, características técnicas y requisitos de uso y también de las incompatibilidades que pueden surgir de su uso combinado. En el desarrollo de software basado en el uso de los llamados componentes COTS

(*Commercial Off-The-Shelf*) resulta crucial el poder determinar si un determinado componente puede reemplazar a otro en una aplicación concreta, o si el comportamiento de dos componentes es compatible y su interoperación es posible.

La noción de compatibilidad que presentamos en esta sección trata de formalizar cómo reconocer si dos roles, especificados mediante agentes en el cálculo π, presentan comportamientos que son conformes o se ajustan uno con otro.

2.3.1 Relación de compatibilidad

Una caracterización formal de la compatibilidad se recoge en la Definición 2.23. A grandes rasgos, diremos que dos roles son compatibles si pueden sincronizarse en al menos una transición (1), si cualquier decisión local en uno de ellos puede ser seguida por el otro (2), y si cualquier par de transiciones *complementarias* lleva a agentes compatibles (3, 4). Intuitivamente, éstas son las condiciones necesarias para asegurar que no se producirá ningún desajuste entre ambos cuando se compongan en paralelo, representando la conexión de los correspondientes componentes.

Para que dos agentes sean compatibles deben ser capaces de sincronizarse en al menos una transición. Esta es una condición necesaria, aunque no suficiente, para la compatibilidad, al impedir que consideremos compatibles agentes como $a(x).\mathbf{0}$ y $\bar{b}y.\mathbf{0}$, que se bloquearían en caso de que los conectásemos uno con otro.

Definición 2.21 *Un agente P* proporciona una entrada *a un agente Q si existen P', Q' tales que*

1. $P \stackrel{\bar{x}y}{\Longrightarrow} P'$ *y* $Q \stackrel{x(z)}{\Longrightarrow} Q'$, *o bien*
2. $P \stackrel{\bar{x}(z)}{\Longrightarrow} P'$ *y* $Q \stackrel{x(z)}{\Longrightarrow} Q'$

Definición 2.22 (Agentes sincronizables) *Dos agentes P y Q son* sincronizables *si P proporciona una entrada a Q o Q proporciona una entrada a P.*

Definición 2.23 (Relación de compatibilidad de base) *Una relación binaria $\mathcal{C}$ definida sobre agentes es una* semi-compatibilidad *si $P\ \mathcal{C}\ Q$ implica*

1. *si P no tiene éxito, entonces P y Q son sincronizables,*
2. *si $P \stackrel{\tau}{\longrightarrow} P'$, entonces $P'\ \mathcal{C}\ Q$,*
3. *si $P \stackrel{x(w)}{\longrightarrow} P'$ y $Q \stackrel{\bar{x}y}{\longrightarrow} Q'$ entonces $P'\{y/w\}\ \mathcal{C}\ Q'$*
4. *si $P \stackrel{x(w)}{\longrightarrow} P'$ y $Q \stackrel{\bar{x}(w)}{\longrightarrow} Q'$ entonces $P'\ \mathcal{C}\ Q'$*

Una relación $\mathcal{C}$ es una compatibilidad *si tanto $\mathcal{C}$ como $\mathcal{C}^{-1}$ son semi-compatibilidades. La compatibilidad de base sobre agentes $\dot{\diamond}$ se define como la relación de compatibilidad más grande.*

Observación 2.24 *Obsérvese que si $P \sim \mathbf{0}$ pero $P \not\equiv \mathbf{0}$ (por ejemplo si $P = (a)a(x).\mathbf{0}$), entonces se tiene que $\forall\, Q\ P \not\dot{\diamond} Q$.*

El diferente tratamiento que en la Definición 2.23 se da a las transiciones silenciosas y no silenciosas tiene que ver con las decisiones globales y locales, tal como muestra el siguiente ejemplo:

Ejemplo 2.25 *Supongamos los agentes:*

$R_1 = a(x).\mathbf{0} + b(y).\mathbf{0}$
$R_2 = \tau.a(x).\mathbf{0} + \tau.b(y).\mathbf{0}$

Aunque ambos presenten las mismas acciones de entrada, estas acciones aparecen como decisiones globales en R_1, mientras que en R_2 el compromiso con una acción determinada, $a(x)$ o $b(y)$ se decide de forma local. Esto ocasiona que agentes que son compatibles con R_1 no lo sean con R_2. Supongamos, por ejemplo,

$\overline{R_2} = \tau.\overline{a}u.\mathbf{0} + \tau.\overline{b}v.\mathbf{0}.$

En este caso se tiene que $R_1 \dot{\diamond} \overline{R_2}$, debido a que el primero es capaz de aceptar cualquier decisión tomada por el segundo, lo que satisface la condición (2) de la Definición 2.23. Sin embargo, no se tiene que $R_2 \dot{\diamond} \overline{R_2}$, debido a que ahora no se satisface dicha condición (2), ya que R_2 puede progresar a $b(y).\mathbf{0}$, y $\overline{R_2}$ a $\overline{a}u.\mathbf{0}$, en ambos casos mediante la realización de una transición silenciosa, y $b(y).\mathbf{0}$ no es compatible de base con $\overline{a}u.\mathbf{0}$, debido a que ni tienen éxito ambos ni son sincronizables.

Las decisiones globales que no aparecen de forma complementaria en ambos agentes se ignoran en la definición de compatibilidad anteriormente expuesta. Supongamos un agente que presente un comportamiento no deseado si se compromete con una determinada acción $\overline{c}$, tal como:

$\overline{R_3} = \overline{a}u.\mathbf{0} + \overline{b}v.\mathbf{0} + \overline{c}\textsc{Error}$

Entonces, se tiene que $R_1 \dot{\diamond} \overline{R_3}$ y $R_2 \dot{\diamond} \overline{R_3}$, ya que la acción c no está prevista en dichos agentes, por lo que dicha transición nunca tendrá lugar al componer $\overline{R_3}$ con R_1 o con R_2. Sin embargo, un agente que pudiese elegir de forma local el comprometerse con esta acción, como:

$\overline{R_4} = \overline{a}u.\mathbf{0} + \overline{b}v.\mathbf{0} + \tau.\overline{c}\textsc{Error}$

no es compatible con ninguno de los agentes citados. □

Definición 2.26 (Agentes compatibles) *P y Q son* compatibles, *lo que se representa como $P \diamond Q$, si $P\sigma \dot{\diamond} Q\sigma$ para toda sustitución σ que respete la distinción definida sobre el conjunto de nombres $fn(P) \cup fn(Q)$.*

Con objeto de comprender por qué motivo utilizamos una distinción entre los nombres $fn(P) \cup fn(Q)$, rechazando así las sustituciones que liguen nombres libres, consideremos el siguiente ejemplo:

Ejemplo 2.27 *Supongamos los roles:*

$P = \bar{a}(x).\bar{x}.\mathbf{0}$
$Q = a(y).y.\mathbf{0} + b(z).\mathbf{0}$

Para ellos se tiene que $P \dot{\diamond} Q$ *para la mayoría de las sustituciones, pero para* $\{a/b\}$ *se tiene que:*

$P\{a/b\} \xrightarrow{\bar{a}(x)} \bar{x}.\mathbf{0}$
$Q\{a/b\} \xrightarrow{a(x)} \mathbf{0}$

y sin embargo, $\bar{x}.\mathbf{0} \not\dot{\diamond} \mathbf{0}$. □

La restricción establecida sobre las sustituciones refleja la situación con los componentes de software reales. Si dos componentes siguen un cierto protocolo en su interacción, es necesario distinguir entre los distintos canales o mensajes usados en dicho protocolo; si no la comunicación sería imposible. Lo mismo se aplica a los roles. Los nombres libres de un rol representan los distintos *nombres* usados en la interacción con otros roles, y estos nombres deben distinguirse unos de otros. Obsérvese que, al conectar varios roles, se restringen sus nombres libres (véase la Definición 2.18), con lo que, de esta forma, se asegura su distinción.

Al igual que la relación de bisimilitud caracterizada por Milner, nuestra definición de compatibilidad implica la cuantificación universal de las sustituciones. Sin embargo, siguiendo una estrategia similar a la recogida en [Sangiorgi, 1993], es posible definir un sistema de transiciones para la relación que sea eficiente y automatizable, lo que permite el desarrollo de herramientas de análisis.

2.3.2 Propiedades de la compatibilidad

La relación de compatibilidad que se ha presentado es simétrica, pero debido a que exige la presencia de acciones complementarias, no satisface varias propiedades comunes como la reflexividad o la transitividad. Incluso aunque hiciésemos abstracción del signo de las acciones no se satisfarían estas propiedades, por lo que no existe ninguna implicación lógica entre compatibilidad y bisimilitud. En efecto, consideremos de nuevo los agentes presentados en el Ejemplo 2.25, donde $R_1 \diamond \overline{R_3}$ a pesar de que ambos presentaban transiciones distintas, pero $R_2 \not\diamond \overline{R_2}$. De hecho, la relación de similitud no se adapta bien a nuestros propósitos, puesto que para hablar de compatibilidad no precisamos que los procesos relacionados se comporten de forma idéntica o encajen perfectamente. Por este motivo, la compatibilidad no ha sido definida con la idea de comparar procesos, sino para disponer de una forma segura y flexible de conectarlos, y esta carencia de propiedades comunes parece razonable. Es decir, mientras que la bisimilitud se define con el objeto de comprobar si dos procesos se simulan uno al otro, la compatibilidad busca la composición 'segura' de procesos.

No obstante, la relación de compatibilidad sí que satisface otras propiedades deseables. En primer lugar, la equivalencia debería preservar la compatibilidad. Supongamos los siguientes agentes:

$P = \mathbf{0}$
$P'(a) = (a)a(x).\mathbf{0}$
$Q = \tau.\mathbf{0}$

Aunque $P \sim P'$ y $P \diamond Q$, se tiene que $P' \not\diamond Q$. La razón es que, desafortunadamente, ni la bisimulación fuerte ni la débil pueden diferenciar entre terminación con éxito y fracaso. Debido a que esta distinción es crucial para nuestros propósitos, hemos definido una relación ligeramente más fina, que denominaremos *simulación congruente en la inacción*, como sigue:

Definición 2.28 *$\mathcal{S}_0$ es una* simulación congruente en la inacción*, si $P\ \mathcal{S}_0\ Q$ satisface las condiciones de la Definición 2.3 y también*

- *Si $P \equiv \mathbf{0}$, entonces $Q \equiv \mathbf{0}$*

Como es lógico, $P\mathcal{S}_0 Q \Longrightarrow P\mathcal{S}Q$. A partir de esta definición, se pueden derivar las de bisimilitud congruente en la inacción fuerte y débil, respectivamente $\sim_0$ y $\approx_0$, y también las de procesos débil y fuertemente bisimilares congruentes en la inacción. Una vez hecho esto, es posible derivar, a partir de la Definición 2.23, que tanto la bisimilitud congruente en la inacción fuerte como la débil preservan la compatibilidad.

Teorema 2.29 *Sean P y Q dos procesos compatibles. Si $P' \approx_0 P$ y $Q' \approx_0 Q$ entonces,*

$$P' \diamond Q'$$

Demostración. Puede derivarse directamente de la Definición 2.23. Sólo es necesario probar que $\diamond_{débil} = \{(P', Q') : \exists\, P, Q$ tales que $P \approx_0 P' \wedge Q \approx_0 Q' \wedge P \diamond Q\}$ es una relación de compatibilidad.

Para ello, y puesto que la compatibilidad es simétrica, basta con probar que si $P \diamond Q$ y $R \approx_0 P$ entonces $R \diamond Q$. En otras palabras, es necesario probar que $\diamond_{débil} = \{(R, Q) : \exists\, P$ tal que $P \approx_0 R \wedge P \diamond Q\}$ es una relación de compatibilidad. Supongamos que $R \diamond_{débil} Q$, entonces comprobamos las condiciones de la Definición 2.23.

1. a) Si R no tiene éxito, debido a que $R \approx_0 P$, entonces P tampoco tiene éxito. En este caso, por $P \diamond Q$ y la Def. 2.23.1 se tiene que P y Q son sincronizables, con lo que $\exists\, \alpha, P', Q'.\ P \stackrel{\alpha}{\Longrightarrow} P'$ y $Q \stackrel{\overline{\alpha}}{\Longrightarrow} Q'$ (donde $\overline{\alpha}$ representa una acción complementaria a α). Puesto que $R \approx_0 P$ se tiene que $\exists\, R'.\ R \stackrel{\alpha}{\Longrightarrow} R'$. Por tanto, R y Q son sincronizables.

   b) Por otro lado, si Q no tiene éxito, entonces de $P \diamond Q$ y la Def 2.23.1 se tiene que $\exists\, \alpha, P', Q'.\ P \stackrel{\alpha}{\Longrightarrow} P'$ y $Q \stackrel{\overline{\alpha}}{\Longrightarrow} Q'$. Puesto que $R \approx_0 P$ se tiene que $\exists\, R'.\ R \stackrel{\alpha}{\Longrightarrow} R'$. Por tanto, R y Q son sincronizables.

2. a) Si $R \stackrel{\tau}{\longrightarrow} R'$, puesto que $R \approx_0 P$, entonces $\exists\, P'$ tal que $P \Longrightarrow P'$ y $R' \approx_0 P'$. Por otra parte, si P' es P (no se lleva a cabo ninguna transición τ), entonces de $R' \approx_0 P'$ y $P' \diamond Q$, se infiere que $R' \diamond_{débil} Q$. Si, por el contrario, $P(\longrightarrow)^+ P'$ (hay al menos una transición τ entre ambos), puesto que $P \diamond Q$, se tiene (por la Def. 2.23.2) que $P' \diamond Q$. De nuevo, de $R' \approx P'$ y $P' \diamond Q$, se infiere que $R' \diamond_{débil} Q$.

   b) Si $Q \stackrel{\tau}{\longrightarrow} Q'$, de $P \diamond Q$ y la Def. 2.23.2 se tiene que $P \diamond Q'$. A partir de esto y también de $R \approx_0 P$, se infiere que $R \diamond_{débil} Q'$.

3. a) Si $R \stackrel{x(w)}{\longrightarrow} R'$ y $Q \stackrel{\bar{x}y}{\longrightarrow} Q'$, puesto que $R \approx_0 P$ entonces $\exists\, P'$ tal que $P \Longrightarrow P'' \stackrel{x(w)}{\longrightarrow} P''' \Longrightarrow P'$ y $R' \approx_0 P'$. Entonces, de $P \diamond Q$ y las Defs. 2.23.2 y 2.23.3 se tiene que $P'' \diamond Q$, $P'''\{y/w\} \diamond Q'$, y $P'\{y/w\} \diamond Q'$. A partir de esto y de $R' \approx_0 P'$ se tiene que $R'\{y/w\} \diamond_{débil} Q'$

b) Si $Q \xrightarrow{x(w)} Q'$ y $R \xrightarrow{\bar{x}y} R'$, puesto que $R \approx_0 P$ entonces $\exists\, P'$ tal que $P \stackrel{\bar{x}y}{\Longrightarrow} P'$ y $R' \approx_0 P'$. Entonces, de las Defs. 2.23.2 y 2.23.3 se infiere que $P' \diamond Q'\{y/w\}$. A partir de esto y de $R' \approx_0 P'$ se tiene que $R' \diamond_{débil} Q'\{y/w\}$.

4. a) Si $R \xrightarrow{x(w)} R'$ y $Q \xrightarrow{\bar{x}(w)} Q'$, puesto que $R \approx_0 P$ entonces $\exists\, P'$ tal que $P \Longrightarrow P'' \xrightarrow{x(w)} P''' \Longrightarrow P'$ y $R' \approx_0 P'$. Entonces, de $P \diamond Q$ y las Defs. 2.23.2 y 2.23.4 se tiene que $P'' \diamond Q$, $P''' \diamond Q'$, y $P' \diamond Q'$. A partir de esto y de $R' \approx_0 P'$ se tiene que $R' \diamond_{débil} Q'$

   b) Si $Q \xrightarrow{x(w)} Q'$ y $R \xrightarrow{\bar{x}(w)} R'$, puesto que $R \approx_0 P$, entonces $\exists\, P'$ tal que $P \stackrel{\bar{x}(w)}{\Longrightarrow} P'$ y $R' \approx_0 P'$. Entonces, de las Defs. 2.23.2 y 2.23.4 se tiene que $P' \diamond Q'$. A partir de esto y de $R' \approx_0 P'$ se tiene que $R' \diamond_{débil} Q'$. ■

La compatibilidad no es una congruencia respecto a todas las construcciones del cálculo π, pero sí se cumplen algunas propiedades interesantes relacionadas con esto. Estas propiedades pueden ser utilizadas para simplificar el análisis de la compatibilidad entre procesos.

Teorema 2.30

a. De $P \diamond Q$ se infiere que $\tau.P \diamond \tau.Q$,

b. De $P\{y/w\} \diamond Q$ se infiere que $x(w).P \diamond \overline{x}y.Q$,

c. De $P \diamond Q$ se infiere que $x(w).P \diamond \overline{x}(w).Q$,

d. De $P_1 \diamond Q$ y $P_2 \diamond Q$ se infiere $P_1 + P_2 \diamond Q$,

e. De $P_1 \diamond Q_1$ y $P_2 \diamond Q_2$ se infiere que $P_1 \mid P_2 \diamond Q_1 \mid Q_2$,
siempre y cuando $fn(P_1) \cap fn(P_2) = \varnothing$ y $fn(Q_1) \cap fn(Q_2) = \varnothing$,

Demostración. Todas ellas pueden demostrarse fácilmente. Como ejemplo, la demostración de la propiedad (d) aparece en el Teorema 2.57. ■

2.3.3 Compatibilidad y composición con éxito

La compatibilidad debe garantizar que no se producirá ningún desajuste durante la interacción de los agentes involucrados. Este es el objetivo de la Proposición 2.31 que figura a continuación, la cual establece que el proceso asociado a la conexión de dos agentes tiene éxito:

Proposición 2.31 *Sean P y Q agentes compatibles. Entonces se tiene que su conexión tiene éxito.*

Demostración. Puede demostrarse directamente a partir de las Definiciones 2.18 y 2.23. Para ello tenemos que demostrar que si $P \diamond Q$ entonces $(fn(P) \cup fn(Q))(P \mid Q)$ tiene éxito. Dado que todos los nombres libres de ambos agentes están restringidos, para abreviar podemos reformular la conexión como $(\mathcal{N})(P \mid Q)$, donde $\mathcal{N}$ contiene cualquier nombre. La prueba se realiza entonces suponiendo que la conexión no tiene éxito, con objeto de encontrar una contradicción.

Supongamos que $P \diamond Q$ pero que $(\mathcal{N})(P \mid Q)$ no tiene éxito. Por tanto, existe un proceso $Fracaso$ tal que $(\mathcal{N})(P \mid Q) \Longrightarrow Fracaso$, con $Fracaso \not\equiv \mathbf{0}$ y $Fracaso \stackrel{\tau}{\not\longrightarrow}$. La demostración se realiza mediante inducción sobre el número n de transiciones τ que llevan a $Fracaso$.

1. *Caso base.* Supongamos primero que $n = 0$. En este caso, de $(\mathcal{N})(P \mid Q) \stackrel{\tau}{\not\longrightarrow}$ se tiene tanto que $P \stackrel{\tau}{\not\longrightarrow}$ como que $Q \stackrel{\tau}{\not\longrightarrow}$. De $(\mathcal{N})(P \mid Q) \not\equiv \mathbf{0}$ se tiene que, o bien $P \not\equiv \mathbf{0}$, o bien $Q \not\equiv \mathbf{0}$. Por tanto, bien P o bien Q no tienen éxito. Dado que $P \diamond Q$, por la Def.2.23.1 se tiene que $\exists\, \alpha \,.\, P \stackrel{\alpha}{\Longrightarrow}$ y $Q \stackrel{\overline{\alpha}}{\Longrightarrow}$ (donde $\overline{\alpha}$ representa una acción complementaria a α). Por tanto, $(\mathcal{N})(P \mid Q) \stackrel{\tau}{\longrightarrow}$, con lo que hemos llegado a una contradicción.

2. *Hipótesis de inducción.* $\forall\, P', Q'$ tales que $P' \diamond Q'$, si $(\mathcal{N})(P' \mid Q')(\stackrel{\tau}{\longrightarrow})^k F$ con $k < n$, entonces bien $F \stackrel{\tau}{\longrightarrow}$ o bien $F \equiv \mathbf{0}$.

3. *Caso general.* Supongamos que $(\mathcal{N})(P \mid Q) \stackrel{\tau}{\longrightarrow} (\mathcal{N})(P' \mid Q')\ (\stackrel{\tau}{\longrightarrow})^{n-1} \mathit{Fracaso}$. Entonces la primera transición τ ha de ser una de las siguientes:

   - $P \stackrel{\tau}{\longrightarrow} P'$. En este caso, debido a que $P \diamond Q$ se tiene que $P' \diamond Q$.
   - $Q \stackrel{\tau}{\longrightarrow} Q'$. En este caso, debido a que $P \diamond Q$ se tiene que $P \diamond Q'$.
   - $P \stackrel{x(w)}{\longrightarrow} P'$ y $Q \stackrel{\bar{x}y}{\longrightarrow} Q'$. En este caso, debido a que $P \diamond Q$ se tiene que $P'\{y/w\} \diamond Q'$.
   - $P \stackrel{x(w)}{\longrightarrow} P'$ y $Q \stackrel{\bar{x}(w)}{\longrightarrow} Q'$. En este caso, debido a que $P \diamond Q$ se tiene que $P' \diamond Q'$.
   - $Q \stackrel{x(w)}{\longrightarrow} Q'$ y $P \stackrel{\bar{x}y}{\longrightarrow} P'$. En este caso, debido a que $P \diamond Q$ se tiene que $P' \diamond Q'\{y/w\}$.
   - $Q \stackrel{x(w)}{\longrightarrow} Q'$ y $P \stackrel{\bar{x}(w)}{\longrightarrow} P'$. En este caso, debido a que $P \diamond Q$ se tiene que $P' \diamond Q'$.

   Por tanto, si $(\mathcal{N})(P \mid Q) \stackrel{\tau}{\longrightarrow} (\mathcal{N})(P' \mid Q')$ se tiene que $P' \diamond Q'$, y por la hipótesis de inducción, se infiere que o bien $\mathit{Fracaso} \stackrel{\tau}{\longrightarrow}$ o bien $\mathit{Fracaso} \equiv \mathbf{0}$. ■

El resultado que viene a continuación va un paso más allá, al mostrar el efecto de combinar un componentes con roles que sean compatibles con los suyos propios.

Teorema 2.32 *Sea Comp un componente correctamente especificado por un conjunto de roles $\{P_i\}_i$, los cuales representan a Comp en su conexión con otros varios componentes $\{Comp_i\}_i$, tal como muestra la Figura 2.8. Sea Q_i el rol que representa respectivamente cada $Comp_i$ en su conexión con Comp. Suponiendo que $\forall\, i\ P_i \diamond Q_i$, entonces se tiene que*

$$\mathit{Comp} \mid \prod_i (\mathit{fn}(Q_i) - \mathit{fn}(P_i))\, Q_i$$

tiene éxito.

Demostración. Se deriva de las Definiciones 2.10, y 2.16. La Definición 2.16 asegura que los nombres libres en $\{P_i\}_i$ son disjuntos, pero algún Q_i puede tener nombres libres adicionales que colisionen con otros nombres en *Comp*, o incluso en algún otro rol de $\{Q_i\}_i$. Sin embargo, estos nombres libres adicionales están restringidos, asegurándose de este modo la independencia de la conexión de *Comp*. En este caso, a partir de $\forall\, i\ P_i \diamond Q_i$ y de las Definiciones 2.10 y 2.16, se deriva el éxito de la composición de *Comp* con los roles $\{Q_i\}_i$. ■

Ejemplo 2.33 *Consideremos de nuevo el sistema Lector/Escritor a que se refiere el Ejemplo 2.17. A partir de la Definición 2.23, es trivial determinar que:*

$\mathit{Almacenamiento}(\mathit{escribir}) \diamond \mathit{Escritor}(\mathit{escribir})$
$\mathit{Recuperación}(\mathit{leer}) \diamond \mathit{Lector}(\mathit{leer})$

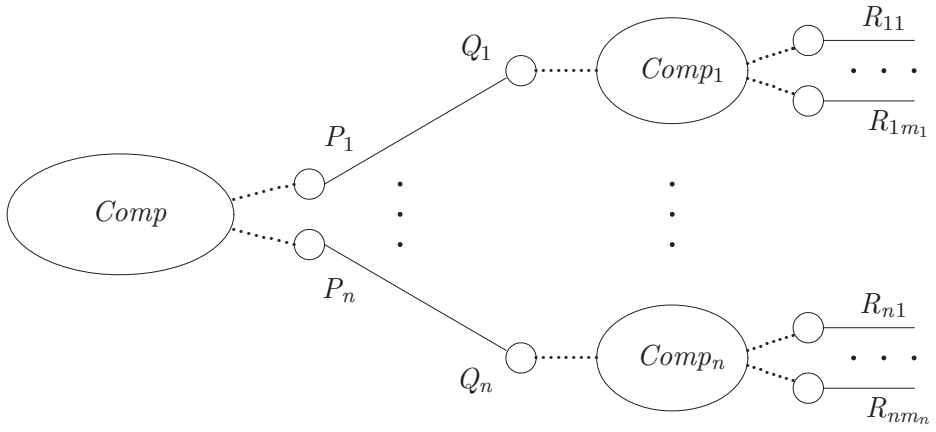


Figura 2.8: Conexiones en las que participa un componente.

Por tanto, por el Teorema 2.32 se tiene que la composición:

$$Escritor(escribir) \mid Variable(leer, escribir) \mid Lector(leer)$$

tiene éxito. Por consiguiente los tres componentes citados pueden ser combinados con éxito para construir un sistema de Lectores/Escritores. □

Obsérvese que el resultado anterior se refiere a la composición de un componente con roles compatibles con los suyos propios, pero no a su composición con los componentes correspondientes a estos roles. De hecho, como vemos de nuevo en la Figura 2.8, aunque hayamos demostrado la compatibilidad de los roles $\{P_i\}_i$ del componente *Comp* con los roles $\{Q_i\}_i$ de los componentes $\{Comp_i\}_i$ a los que está conectado, no podemos considerar de forma aislada la composición paralela de los citados componentes:

$$Comp \mid \prod_i (fn(Q_i) - fn(P_i))\, Comp_i$$

puesto que estos componentes $\{Comp_i\}_i$ estarán, por lo general, conectados a otros componentes del sistema a través de otros de sus roles (representados en la figura como $\{R_{ij}\}_{ij}$), de forma que necesitarán también de estos otros componentes para evolucionar.

Ejemplo 2.34 *Consideremos el sistema que muestra la Figura 2.9, formado por cuatro emisores, tres concentradores y un receptor. Es trivial comprobar que las conexiones de este sistema son todas entre pares de roles compatibles, por lo que, en aplicación del Teorema 2.32, tendríamos, por ejemplo, que para el último de los concentradores, el agente formado por la composición paralela de este* Concentrador *con los roles de los componentes a los que está conectado:*

$$\begin{aligned} Concentrador(s_1, s_2, s) &\mid Emisor(s_1) \\ &\mid Emisor(s_2) \\ &\mid Receptor(s) \end{aligned}$$

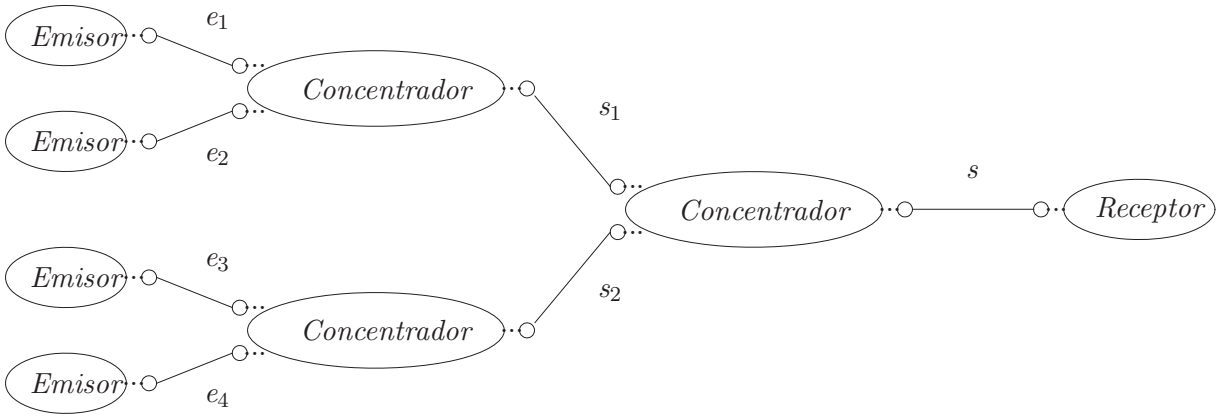


Figura 2.9: Sistema formado por varios emisores y concentradores y un receptor.

tiene éxito. Sin embargo, también es evidente que la composición de los componentes correspondientes:

$$\begin{array}{ll} Concentrador(s_1, s_2, s) & \mid Concentrador(e_1, e_2, s_1) \\ & \mid Concentrador(e_3, e_4, s_2) \\ & \mid Receptor(s) \end{array}$$

no lo tiene, puesto que faltan los componentes emisores, que son quienes ponen en marcha el sistema. □

Incluso aunque considerásemos un sistema en su totalidad, el análisis local de la compatibilidad en cada una de las conexiones tampoco puede demostrar que el sistema en su conjunto tenga éxito, ya que se puede producir un bloqueo a partir de la interacción global de un conjunto de componentes cuyos roles sean compatibles. Veamos esto último por medio de otro ejemplo.

Ejemplo 2.35 *Supongamos que conectamos circularmente, tal como indica la Figura 2.10, dos componentes* Traductor *como los descritos en el Ejemplo 2.12.*

En esta situación, si bien el agente:

$$\begin{array}{ll} Traductor(e_1, e_2, s_1, s_2) & \mid Salida(e_1, e_2) \\ & \mid Entrada(s_1, s_2) \end{array}$$

tiene éxito, es obvio que:

$$Traductor(e_1, e_2, s_1, s_2) \mid Traductor(s_1, s_2, e_1, e_2)$$

se bloquea. □

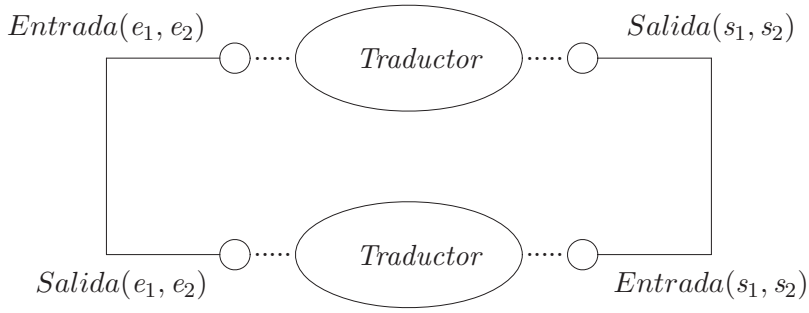


Figura 2.10: Sistema formado por dos traductores conectados circularmente.

No obstante los ejemplos anteriores, debemos recordar que nuestro objetivo no es demostrar si un determinado sistema software está libre de bloqueos (lo que no es posible determinar simplemente a partir del análisis local de sus interfaces, sino que requeriría trabajar con la especificación completa de los componentes). Nuestro propósito es asegurar que no se producirá un fallo o *caída* del mismo debido a un desajuste de comportamiento entre las interfaces de los componentes interconectados. Consideramos esta condición suficiente para comprobar la composicionalidad de un determinado sistema a partir de sus componentes integrantes, así como la posibilidad de reutilizar un determinado componente en un sistema distinto a aquél para el que fue desarrollado originalmente.

Llegados a este punto, podemos definir una *arquitectura composicional*, formada por la composición de varios componentes, como un conjunto de conexiones entre pares compatibles de roles de dichos componentes. Esta definición nos permite la extensión de los resultados de los Teoremas 2.31 y 2.32 a las arquitecturas.

Definición 2.36 (Arquitectura composicional) *Supongamos una arquitectura binaria* Ψ *según las condiciones de la Definición 2.19. Diremos que* Ψ *es una* arquitectura composicional *si* $\forall\, P, Q \in \mathcal{R}oles$*, tales que* $\{P, Q\} \in \Psi$ *se tiene que* $P \diamond Q$*.*

Proposición 2.37 *Sea* Ψ *una arquitectura composicional. Entonces* Ψ *tiene éxito.*

Demostración. Se deriva inmediatamente de la Definición 2.36 y el Teorema 2.31. ■

Según acabamos de ver, la composicionalidad de una arquitectura se determina mediante la comprobación de la compatibilidad de las conexiones entre los roles de sus componentes. Para simplificar, hemos definido una arquitectura como un conjunto de conexiones entre *pares* de roles, ciñéndonos al uso de relaciones binarias. Sin embargo, podemos considerar una definición más general, en la que las conexiones involucren a más de dos roles. Este será el objetivo de la próxima sección.

2.3.4 Compatibilidad de grupos de roles

La relación de compatibilidad de la Definición 2.23 se refiere únicamente a pares de roles. No obstante, la compatibilidad puede extenderse a grupos de más de dos roles, permitiendo de esta forma el análisis de conexiones más complejas, tal como las contempladas en la Definición 2.18.

Definición 2.38 (Conjunto de agentes sincronizables) *Un conjunto de agentes $\mathcal{P}$ es* sincronizable *si $\exists\, P, Q \in \mathcal{P}$ tales que P proporciona una entrada a Q.*

Definición 2.39 (Compatibilidad de base de un conjunto de agentes) *Un conjunto de agentes $\mathcal{P}$ es* compatible de base *si*

1. *si $\exists\, P \in \mathcal{P}$ tal que P no tiene éxito entonces $\mathcal{P}$ es un conjunto de agentes sincronizables,*
2. *si $\exists\, P \in \mathcal{P}$ tal que $P \xrightarrow{\tau} P'$ entonces $(\mathcal{P} - \{P\}) \cup \{P'\}$ es compatible de base,*
3. *si $\exists\, P, Q \in \mathcal{P}$ tales que $P \xrightarrow{x(w)} P'$ y $Q \xrightarrow{\bar{x}y} Q'$ entonces $(\mathcal{P} - \{P, Q\}) \cup \{P'\{y/w\}, Q'\}$ es compatible de base,*
4. *si $\exists\, P, Q \in \mathcal{P}$ tales que $P \xrightarrow{x(w)} P'$ y $Q \xrightarrow{\bar{x}(w)} Q'$ entonces $(\mathcal{P} - \{P, Q\}) \cup \{P', Q'\}$ es compatible de base.*

Definición 2.40 (Compatibilidad de un conjunto de agentes) *Un conjunto de agentes $\mathcal{P} = \{P_i\}_i$ es* compatible *si $\mathcal{P}\sigma = \{P_i\sigma\}_i$ es compatible de base para cualquier sustitución σ que respete la distinción $\cup_i fn(P_i)$.*

Proposición 2.41 *Sea $\mathcal{P}$ un conjunto de agentes compatibles. Entonces, su conexión tiene éxito.*

Demostración. Puede demostrarse directamente a partir de las Definiciones 2.18 y 2.39. ■

Ejemplo 2.42 (Lectores y Escritor) *Supongamos un sistema de Lectores/Escritores más general que aquél al que hacía referencia el Ejemplo 2.17, en el que un conjunto de procesos lectores acceden a los datos generados por un proceso escritor a través de una variable compartida por todos ellos.*

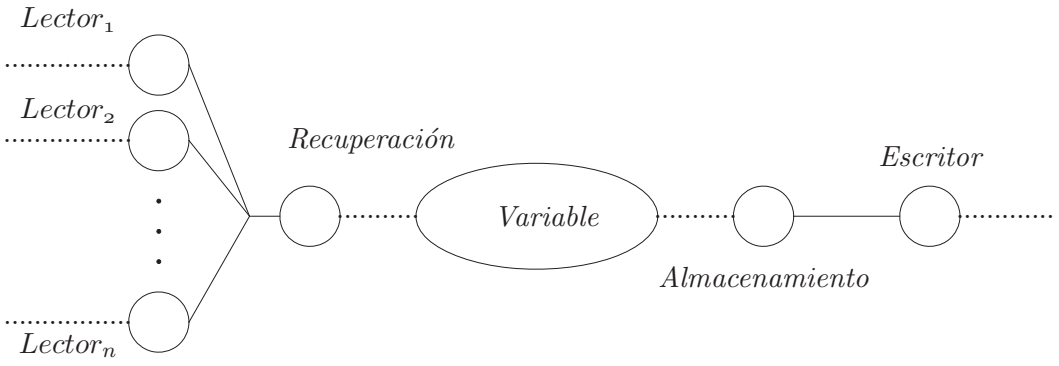


Figura 2.11: Sistema de Lectores y Escritor.

En esta situación tenemos dos conexiones entre roles, tal como describe la Figura 2.11. Por un lado, la conexión entre un rol Almacenamiento *del componente* Variable *y un rol* Escritor

del proceso que escribe en dicha variable, y por el otro, una conexión múltiple entre un rol Recuperación *del componente* Variable *y un número indeterminado de roles* Lector *de los procesos que consultan dicha variable. Refiriéndonos a esta última conexión, mediante la Definición 2.39 trataremos de determinar si estos agentes son compatibles.*

Comprobación. *Aún cuando no sepamos el número de lectores en nuestro sistema, podemos comprobar que la conexión de los agentes:*

$$\{\ Lector_1(leer), Lector_2(leer), \ldots, Lector_n(leer),\ Recuperación(leer)\ \}$$

(cuya especificación figura en el Ejemplo 2.17), constituye un conjunto de agentes compatible. Para ello verificamos el cumplimiento de las condiciones en la Definición 2.39:

1. *Ninguno de los agentes, considerados por separado, tiene éxito. No obstante, resulta evidente, puesto que realizan acciones de entrada y salida a través del mismo enlace, que se trata de agentes sincronizables.*

2. *No es aplicable.*

3. *No es aplicable.*

4. *Se puede aplicar, para cada agente $Lector_i$, en el siguiente caso:*

   $$Recuperación(leer) \xrightarrow{\overline{leer}(valor)} Recuperación(leer)$$

   $$Lector_i(leer) \xrightarrow{leer(valor)} Lector_i(leer)$$

   Independientemente del Lector elegido, llegamos de nuevo al conjunto de agentes original, por lo que damos por finalizada la comprobación, concluyendo que se trata de un conjunto de roles compatibles. ■

Por tanto, la conexión de todos estos roles tendrá éxito, con lo que el proceso:

$$Recuperación(leer)\ \mid\ \prod_i Lector_i(leer)$$

que representa a dicha conexión, estará libre de bloqueos. □

La relación de compatibilidad entre roles definida en esta sección establece las condiciones para poder realizar con éxito la composición de los correspondientes componentes. Sin embargo, con objeto de fomentar la reutilización de tanto de los componentes como de la arquitectura, sería muy interesante disponer de una forma de comprobar si un determinado componente ya existente puede ser utilizado en cualquier contexto en el que aparece otro componente. Este será el objetivo de la sección siguiente.

2.4 Herencia y extensión de comportamiento

La noción de reemplazabilidad entre componentes a la que acabamos de hacer referencia está muy relacionada con los conceptos de herencia, refinamiento y polimorfismo presentes en el paradigma orientado a objetos.

En efecto, en orientación a objetos, la herencia se refiere a una relación entre clases de objetos mediante la cual una clase *vástago* hereda las propiedades declaradas por su *progenitor*, a la vez que añade sus propias características. Las propiedades heredadas pueden ser redefinidas, normalmente bajo ciertas restricciones. Aunque no se puede asegurar que el tipo de la clase vástago sea un subtipo del de su progenitor [Cook et al., 1990], la herencia es una precondición natural del polimorfismo, permitiendo el reemplazamiento dinámico de un objeto por una versión derivada o refinada del mismo, a la vez que fomenta tanto la reutilización como el diseño incremental.

Bajo nuestro enfoque, los roles representan el comportamiento de los componentes correspondientes, es decir, indican cómo reaccionan éstos ante los estímulos externos. Así, un agente que represente a un rol vástago debe mantener en cierta medida el comportamiento de sus progenitores, con objeto de asegurar que el componente progenitor pueda ser reemplazado por una versión derivada manteniendo la compatibilidad. Por este motivo, debemos establecer una serie de restricciones, a la hora de redefinir y añadir comportamiento en los roles derivados, que lleven a que el comportamiento especificado por estos últimos sea más previsible que el de sus respectivos progenitores, mediante la menor toma de decisiones locales (y llamamos a esto *herencia de roles*), a la vez que puedan ofrecer nuevo comportamiento globalmente decidido (lo que denominamos *extensión de roles*).

La relación de herencia entre agentes en el cálculo π que presentaremos a continuación formaliza las ideas que acabamos de exponer, al estar definida con el requisito de que preserve la compatibilidad, permitiendo de esta forma el reemplazamiento en cualquier arquitectura de un componente por una versión derivada del mismo cuyos roles hereden de aquéllos del primero.

2.4.1 Relación de herencia

Una relación de herencia entre procesos que preserve la compatibilidad implica el cumplimiento de varias condiciones, relativas a la herencia del comportamiento del progenitor, su redefinición y su extensión. Por este motivo, presentaremos la relación en varias etapas.

Definición 2.43 (Herencia con mantenimiento de la semántica) *Una relación binaria $\mathcal{H}$ sobre agentes es una relación de* herencia con mantenimiento de la semántica *si $R\mathcal{H}P$ implica que*

1. *si* $P \xrightarrow{\bar{x}y} P'$ *entonces* $\exists\, R'.\ R \xrightarrow{\bar{x}y} R'$ *y* $R'\mathcal{H}P'$
2. *si* $P \xrightarrow{x(y)} P'$, *donde* $y \notin n(P,R)$, *entonces* $\exists\, R'.\ R \xrightarrow{x(y)} R'$ *y* $\forall\, w, R'\{w/y\}\mathcal{H}P'\{w/y\}$
3. *si* $P \xrightarrow{\bar{x}(y)} P'$, *donde* $y \notin n(P,R)$, *entonces* $\exists\, R'.\ R \xrightarrow{\bar{x}(y)} R'$ *y* $R'\mathcal{H}P'$

El mantenimiento de la semántica exige que cualquier comportamiento globalmente decidido en el agente progenitor esté también presente en el vástago. Sin embargo, no se imponen condiciones sobre las transiciones τ, por lo que las decisiones locales pueden ser convertidas en globales o incluso suprimidas por el agente vástago.

Con el propósito de aclarar las condiciones de la definición precedente, definamos $P\diamond = \{Q \ : \ P \diamond Q\}$ como el conjunto de procesos que son compatibles con un determinado agente P (indistintamente por la derecha o por la izquierda, por ser $\diamond$ simétrica), y supongamos un agente R tal que $R\mathcal{HP}$. En este caso, la Definición 2.43 implica que cualquier proceso de $P\diamond$ que tome una decisión sobre comportamientos descritos mediante decisiones globales en P, será también compatible con R, mientras que algunos procesos que no están en $P\diamond$ estarán en $R\diamond$, ya que algunos de los requisitos de P (es decir, algunas de sus decisiones locales) podrán haber sido relajadas o suprimidas por R. De ahí que $P\diamond \subseteq R\diamond$.

Ejemplo 2.44 *Supongamos el agente:*

$$P(a) = a(x).\mathbf{0}$$

Debido a que P ofrece la acción $a(x)$ como una decisión global, algún agente $Q \in P\diamond$, puede comprometerse a la acción complementaria por medio de una decisión local, como por ejemplo:

$$Q(a,b) = \tau.\bar{a}(u).\mathbf{0} + \bar{b}(v).\mathbf{0}$$

Según esto, cualquier agente que quiera heredar a P debe preservar la acción $a(x)$ como una decisión global. Sea por ejemplo:

$$R(a,c) = a(x).\mathbf{0} + c(z).\mathbf{0}$$

atendiendo a la Definición 2.43, se tiene que $R\mathcal{H}P$ y también que $R \dot{\diamond} Q$. Por otra parte, agentes como:

$$S(a,c) = \tau.\bar{a}(u).\mathbf{0} + \tau.\bar{c}(w).\mathbf{0}$$

que no eran compatibles con P son ahora compatibles con R. □

La herencia con mantenimiento de la semántica es una condición necesaria pero no suficiente para la herencia, como muestra el siguiente ejemplo:

Ejemplo 2.45 *Supongamos los agentes:*

$$P(a,b) = a(x).\mathbf{0} + \tau.b(y).\mathbf{0}$$

$$Q(b,c) = \bar{b}(u).\mathbf{0} + \bar{c}(v).v.\mathbf{0}$$

$$R(a,b,c) = a(x).\mathbf{0} + b(y).\mathbf{0} + c(z).\mathbf{0}$$

Atendiendo a la Definición 2.43, se tiene que $P \dot{\diamond} Q$, y también que $R\mathcal{H}P$. Sin embargo, $R \not\dot{\diamond} Q$, ya que $R \xrightarrow{c(v)} \mathbf{0}$, $Q \xrightarrow{\bar{c}(v)} v.\mathbf{0}$, y $\mathbf{0} \not\diamond v.\mathbf{0}$. □

Por tanto, es necesario imponer condiciones adicionales a la herencia con mantenimiento de la semántica, cuya definición debe ser extendida como sigue:

Definición 2.46 (Herencia no extensible) *Una relación binaria $\mathcal{H}$ sobre agentes es una relación de* herencia no extensible *si $R\mathcal{H}P$ implica las condiciones de la Definición 2.43, y también:*

1. *si $R \xrightarrow{\tau} R'$ entonces $\exists\, P'.\ P \xrightarrow{\tau} P'$ y $R'\mathcal{H}P'$*
2. *si $R \xrightarrow{\bar{x}y} R'$ entonces $\exists\, P'.\ P \Longrightarrow\!\xrightarrow{\bar{x}y} P'$ y $R'\mathcal{H}P'$*
3. *si $R \xrightarrow{x(y)} R'$ y $y \notin n(P,R)$ entonces $\exists\, P'.\ P \Longrightarrow\!\xrightarrow{x(y)} P'$ y $\forall\, w, R'\{w/y\}\mathcal{H}P'\{w/y\}$*
4. *si $R \xrightarrow{\bar{x}(y)} R'$ y $y \notin n(P,R)$ entonces $\exists\, P'.\ P \Longrightarrow\!\xrightarrow{\bar{x}(y)} P'$ y $R'\mathcal{H}P'$*

Las condiciones anteriores exigen que el agente vástago R no extienda a su progenitor P mediante el ofrecimiento de nuevos comportamientos, ya sea por medio de decisiones locales o globales (obsérvese sin embargo, que el rol vástago puede haber convertido algunas de las decisiones locales del progenitor en globales). Como se ha visto en el Ejemplo 2.45, la razón para esta restricción de la extensión de comportamiento se debe a que cualquier nueva transición en R puede sincronizar con una transición complementaria en un determinado $Q \in P\diamond$, mientras que esta transición de Q no fue tenida en cuenta al analizar la compatibilidad de P y Q, debido a que la Definición 2.23 se refiere únicamente a las transiciones que son complementarias en ambos agentes. Obsérvese que con la nueva definición, para los agentes del Ejemplo 2.45 se tiene que R no hereda de P.

Como se puede apreciar, con objeto de mantener la compatibilidad nuestra definición de herencia debe ser muy restrictiva. Sin embargo, estas restricciones pueden ser superadas en la Definición 2.55 sobre extensión de agentes.

De nuevo, la herencia no extensible es una condición necesaria para la herencia, pero se requieren aún dos condiciones adicionales.

Definición 2.47 (Relación de herencia) *Una relación binaria $\mathcal{H}$ sobre agentes es una relación de* herencia *si $R\mathcal{H}P$ implica las condiciones de las Definiciones 2.43 y 2.46, y también:*

1. *si $P \equiv \mathbf{0}$ entonces $R \equiv \mathbf{0}$*
2. *si $P \xrightarrow{\tau}$ entonces $\exists\, P', R'.\ P \xrightarrow{\tau} P'$ y $R \Longrightarrow R'$ y $R'\mathcal{H}P'$*

La herencia de agentes $\dot{\rhd}$ se define como la relación de herencia más grande.

La primera condición determina qué agentes heredan de la inacción. La segunda es menos intuitiva. Se refiere a las transiciones τ del agente progenitor (las cuales no fueron tenidas en cuenta en la Definición 2.43), e indica que al menos una de ellas debe ser heredada por el rol vástago. Esta es una condición necesaria para mantener a R sincronizable con cualquier agente en $P\diamond$, ya que sólo las transiciones complementarias a las decisiones locales de P son necesarias en cualquier agente $Q \in P\diamond$.

Ejemplo 2.48 *Supongamos ahora los agentes:*

$P(a,b) = a(x).\mathbf{0} + \tau.b(y).\mathbf{0}$

$R(a) = a(x).\mathbf{0}$

que satisfacen todas las condiciones de la herencia excepto la citada de la Definición 2.47.2. Se tiene que $R \dot{\diamond} Q$ *para la mayoría de los* $Q \in (\diamond P)$*, como por ejemplo:*

$$Q(a,b) = \overline{a}(u).\mathbf{0} + \overline{b}(v).\mathbf{0}$$

pero para

$$Q'(b) = \overline{b}(v).\mathbf{0}$$

se tiene que $P \dot{\diamond} Q'$ *pero* $R \not\dot{\diamond} Q'$*. Por el contrario, para:*

$$P'(a,b) = \tau.a(x).\mathbf{0} + \tau.b(y).\mathbf{0}$$

se tiene que $R\mathcal{H}P'$ *(satisfaciendo ahora todas las condiciones de la herencia),* $P' \dot{\diamond} Q$*, y también que* $R \dot{\diamond} Q$*.* □

Definición 2.49 (Herencia de comportamiento) R hereda *a* P*, lo que se representa como* $R \rhd P$*, si* $R\sigma \dot{\rhd} P\sigma$ *para cualquier sustitución* σ *que respete la distinción* $fn(P) \cup fn(R)$*.*

Como se puede comprobar fácilmente, la relación de herencia que acabamos de presentar es reflexiva, antisimétrica y transitiva, por lo que se trata de una relación de orden parcial, definida sobre el conjunto de agentes del cálculo π.

Al introducir la noción de herencia hacíamos referencia a que el objetivo era establecer una relación entre pares de agentes, que hemos denominado vástago y progenitor, que determinase aquellos casos en los que el vástago relajaba las exigencias de su progenitor de forma que era compatible con un conjunto más grande de agentes. Una vez definida formalmente la herencia de comportamiento, podemos presentar un ejemplo que ilustre este aspecto.

Ejemplo 2.50 (Espacios de tuplas) *Volviendo al ejemplo del espacio de tuplas al que hace referencia la Sección 2.1.8, podemos especificar el rol que muestra el comportamiento del componente* Espacio*, según este comportamiento es observado por los procesos que lo utilizan:*

$$\begin{aligned} RolEspacio(in,out,rd) = {} & out(tupla\ datos).RolEspacio(in,out,rd) \\ & + rd(tupla\ return).(\ (\ \tau.\overline{return}(datos).\mathbf{0} + \tau.\mathbf{0}\ ) \\ & \qquad\qquad \mid\ RolEspacio(in,out,rd)\ ) \\ & + in(tupla\ return).(\ (\ \tau.\overline{return}(datos).\mathbf{0} + \tau.\mathbf{0}\ ) \\ & \qquad\qquad \mid\ RolEspacio(in,out,rd)\ ) \end{aligned}$$

El agente RolEspacio *describe la interfaz del espacio de tuplas. En este rol se hace abstracción del comportamiento interno del espacio, que no es observable desde su entorno. Así, cuando un proceso intenta leer una tupla (ya sea mediante el envío de un mensaje a través del enlace* rd *o a través de* in*), el comportamiento descrito en el rol indica que el espacio de tuplas puede responder a la petición de lectura enviando los datos de la tupla a través el enlace de retorno (lo que se especifica mediante la alternativa* $\tau.\overline{return}(datos).\mathbf{0}$*), o bien puede ignorar indefinidamente dicha petición, debido a que la tupla solicitada no se encuentre en el espacio (lo que se modela a mediante la alternativa* $\tau.\mathbf{0}$*). Estamos entonces ante una decisión local del espacio, que no puede ser prevista desde el punto de vista del proceso que lo utiliza.*

Especifiquemos ahora el comportamiento de un proceso que se sirva del espacio de tuplas para coordinarse con otros. El rol RolProceso *describe la forma general de este tipo de procesos, que realizan peticiones de lectura y escritura de tuplas en un orden no determinado.*

$$\begin{aligned} RolProceso(in, out, rd) = \\ &\tau.\overline{in}(tupla\ return).return(datos).RolProceso(in, out, rd) \\ &+ \tau.\overline{out}(tupla\ datos).RolProceso(in, out, rd) \\ &+ \tau.\overline{rd}(tupla\ return).return(datos).RolProceso(in, out, rd) \end{aligned}$$

Si ahora comprobamos la compatibilidad entre ambos roles, haciendo uso de la Definición 2.23, llegaríamos a la conclusión de que:

$$RolProceso(in, out, rd) \not\diamond RolEspacio(in, out, rd)$$

puesto que tras la sincronización de ambos roles al realizar, por ejemplo, las acciones $\tau.\overline{in}(tupla\ return)$ *e* $in(tupla\ return)$*, respectivamente, el rol que representa al proceso queda esperando la ocurrencia de una acción de entrada a través el enlace de retorno —*$return(datos)$*— mientras que el* RolEspacio *puede optar simplemente por ignorar la petición, realizando una transición silenciosa* τ*, con lo que no se cumplen las condiciones de la Definición 2.23.*

No obstante, podemos encontrar una variante de nuestro espacio de tuplas que sea compatible con el comportamiento que describe RolProceso*. En efecto, existen extensiones de Linda que incorporan operaciones de lectura* in *y* rd *no bloqueantes, y que devuelven un valor de retorno especial (que aquí representaremos mediante la constante* Null*) en caso de que la lectura no haya podido realizarse por ausencia de la tupla. La especificación (incompleta) del rol que modela el comportamiento de un espacio de tuplas con operaciones de lectura no bloqueantes, sería la siguiente:*

$$\begin{aligned} RolEspacio'(in, out, rd) = \\ &in(tupla\ return).(\ (\ \tau.\overline{return}(datos).\mathbf{0} + \tau.\overline{return}\ \textsc{Null}.\mathbf{0}\ ) \\ &\qquad\qquad\qquad\qquad \mid\ RolEspacio'(in, out, rd)\ ) \\ &+ \ldots \end{aligned}$$

lo que puede simplificarse como:

$$\begin{aligned} RolEspacio'(in, out, rd) = \\ &in(tupla\ return).(\ \overline{return}(datos).\mathbf{0} \mid\ RolEspacio'(in, out, rd)\ ) \\ &+ \ldots \end{aligned}$$

En esta situación, y haciendo uso de nuevo de la Definición 2.23, podemos determinar que ahora:

$$RolProceso(in, out, rd) \diamond RolEspacio'(in, out, rd)$$

es decir, que hemos eliminado las exigencias del agente RolEspacio *que impedían que fuese compatible con* RolProceso*, siendo por tanto* RolEspacio′ *menos restrictivo que* RolEspacio*, al ser el primero compatible con un conjunto mayor de roles. Si analizamos ahora las dos versiones del rol del espacio de componentes haciendo uso para ello de la relación de herencia de comportamiento de la Definición 2.47, llegamos a que:*

$$RolEspacio'(in, out, rd) \rhd RolEspacio(in, out, rd)$$

que indica que la segunda versión del rol del espacio de tuplas (la correspondiente a las operaciones de lectura no bloqueantes) es heredera de la primera que presentamos. La Figura 2.12 resume las relaciones entre los roles utilizados en este ejemplo. □

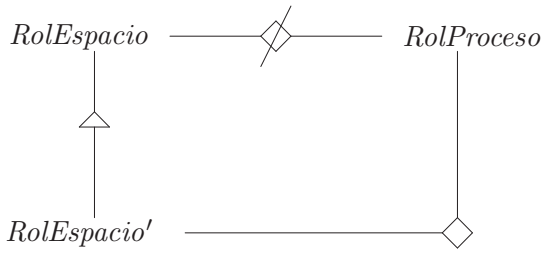


Figura 2.12: Relaciones entre los roles del Ejemplo 2.50.

El ejemplo que acabamos de presentar apunta a que la relación de herencia está definida con objeto de hacer al rol vástago compatible con un conjunto mayor de roles que los que eran compatibles con su progenitor. Para ello, no sólo debe seguir siendo compatible con aquellos roles con los que su progenitor lo era, sino que debe serlo también con roles con los que su progenitor no era compatible. El resultado que presentamos a continuación formaliza esta idea, al demostrar que la herencia de comportamiento preserva la compatibilidad. Para demostrar el Teorema 2.52 es preciso hacer uso de una condición adicional, y por esta razón, rechazamos los roles que sean *semánticamente divergentes*, es decir que realicen infinitas computaciones internas (representadas mediante transiciones τ). Sin embargo, esto no debe ser considerado como una restricción, puesto que estos roles *patológicos* serán típicamente de la forma $P = \ldots + \tau.P + \ldots$, y siempre podremos encontrar un agente P' tal que $P' \approx P$, donde P' no presente una traza infinita de transiciones τ y P' satisfaga la Definición 2.16, como muestra el siguiente ejemplo:

Ejemplo 2.51 *Supongamos el componente:*

$$Comp(a, b) = a(x).Comp(a, b) + b(y).\mathbf{0}$$

con sus roles

$$Rol_1(a) = a(x).Rol_1(a) + \tau.\mathbf{0}$$

$$Rol_2(b) = \tau.Rol_2(b) + b(y).\mathbf{0}$$

obtenidos de acuerdo a las Definiciones 2.10 y 2.16.

Como podemos ver, Rol_2 presenta una traza infinita de transiciones τ, pero podemos encontrar una versión débilmente bisimilar:

$$Rol'_2(b) = b(y).\mathbf{0}$$

sin trazas infinitivas de transiciones τ, tal que $Rol_2' \approx Rol_2$, donde $\{Rol_1, Rol_2'\}$ es también un conjunto correcto de roles de Comp. □

Teorema 2.52 *Sean P y Q dos procesos, donde P no presente ninguna traza infinita de transiciones τ. Sea $P \diamond Q$ y sea $R \rhd P$. Entonces se tiene que $R \diamond Q$.*

Demostración. Puede ser derivada a partir de las Definiciones 2.23 y 2.47. Basta con probar que $\diamond_{her} = \{(R, Q) : \exists P \,.\, P \diamond Q \wedge R \rhd P\}$ es una relación de compatibilidad.

Supongamos que $R \diamond_{her} Q$, y comprobemos las condiciones de la Definición 2.23.

1. a) Si R no tiene éxito, entonces por la Def. 2.14 se tiene que $\exists R'$ tal que $R \Longrightarrow R'$, donde $R' \not\equiv \mathbf{0}$ y $R' \stackrel{\tau}{\not\longrightarrow}$.

   Supongamos primero que R' es R. Por $R \rhd P$, $R \not\equiv \mathbf{0}$, y $R' \stackrel{\tau}{\not\longrightarrow}$, a partir de la Def. 2.47 se infiere que $P \not\equiv \mathbf{0}$ y $P \stackrel{\tau}{\not\longrightarrow}$, es decir, P no tiene éxito. Entonces, por $P \diamond Q$ y la Def. 2.23.1 se tiene que P y Q son sincronizables. Por tanto, $\exists \alpha$ tal que $P \stackrel{\alpha}{\longrightarrow} P'$ y $Q \stackrel{\overline{\alpha}}{\Longrightarrow} Q'$, (donde $\overline{\alpha}$ representa una acción complementaria a α). Ahora, por $P \stackrel{\alpha}{\longrightarrow} P'$ y la Def. 2.43 se tiene que $R \stackrel{\alpha}{\longrightarrow} R'$. Por tanto, R y Q son sincronizables.

   Supongamos ahora que R' no es R, es decir, $R(\stackrel{\tau}{\longrightarrow})^n R'$, con $n > 0$. Por $R \rhd P$ y la Def. 2.46.1 se tiene que $\exists P' \,.\, P(\stackrel{\tau}{\longrightarrow})^n P'$ y $R' \rhd P'$. De nuevo, $R' \not\equiv \mathbf{0}$, $R' \stackrel{\tau}{\not\longrightarrow}$ y la Def. 2.47 implican que $P' \not\equiv \mathbf{0}$ y $P' \stackrel{\tau}{\not\longrightarrow}$. Como $P \diamond Q$, por la Def. 2.23.2 se tiene que $P' \diamond Q$, y como P no tiene éxito, por la Def. 2.23.1, $\exists \alpha$ tal que $P' \stackrel{\alpha}{\longrightarrow} P''$ y $Q \stackrel{\overline{\alpha}}{\Longrightarrow} Q'$, (sin transiciones τ desde P'). De esto último, de $R' \rhd P'$, y de la Def. 2.43, se infiere que $R' \stackrel{\alpha}{\longrightarrow} R''$. Por tanto, $R \stackrel{\alpha}{\Longrightarrow} R''$, y R y Q son sincronizables.

   b) Si Q no tiene éxito, entonces por la Def. 2.14 $\exists Q'$ tal que $Q \Longrightarrow Q'$, donde $Q' \not\equiv \mathbf{0}$ y $Q' \stackrel{\tau}{\not\longrightarrow}$. Dado que $P \diamond Q$, por la Def. 2.23.2 se tiene que $P \diamond Q'$, y como Q' no tiene éxito, se tiene que P y Q' son sincronizables. Por tanto, $\exists \alpha$ tal que $P \stackrel{\alpha}{\Longrightarrow} P''$ y $Q' \stackrel{\alpha}{\longrightarrow} Q''$ (sin transiciones τ desde Q').

   Supongamos primero que $P \stackrel{\alpha}{\Longrightarrow} P''$ es $P \stackrel{\alpha}{\longrightarrow} P''$. En este caso, por la Def. 2.43 se tiene que $R \stackrel{\alpha}{\longrightarrow} R''$. Por tanto, R y Q' son sincronizables, y también R y Q son sincronizables.

   Supongamos ahora que $P \stackrel{\tau}{\longrightarrow} \stackrel{\alpha}{\Longrightarrow} P''$ (es decir, hay al menos una transición τ entre P y P''). Por el Lema 2.53 que figura más abajo, se tiene que $\exists P', R' \,.\, P \Longrightarrow P'$, $R \Longrightarrow R', R' \rhd P'$ y $P' \stackrel{\tau}{\not\longrightarrow}$. Entonces, por $P \diamond Q'$ se tiene que $P' \diamond Q'$. Dado que Q' no tiene éxito, se tiene que $\exists \beta$ tal que $P' \stackrel{\beta}{\longrightarrow}, Q' \stackrel{\beta}{\longrightarrow}$ (sin transiciones τ). Entonces, como $R' \rhd P'$, por la Def. 2.43 se tiene que $R' \stackrel{\beta}{\longrightarrow}$. Por tanto, se tiene que R' y Q' son sincronizables, y consiguientemente también lo son R y Q.

2. a) Si $R \stackrel{\tau}{\longrightarrow} R'$ entonces, por $R \rhd P$ y la Def. 2.46.1, se tiene que $\exists P'$ tal que $P \stackrel{\tau}{\longrightarrow} P'$ y $R' \rhd P'$. Debido a que $P \diamond Q$, por la Def. 2.23.2 se tiene que $P' \diamond Q$. Por tanto, $R' \diamond_{her} Q$.

   b) Si $Q \stackrel{\tau}{\longrightarrow} Q'$ entonces, por $P \diamond Q$ y Def. 2.23.2 se tiene que $P \diamond Q'$. De lo anterior, y de $R \rhd P$, se infiere que $R \diamond_{her} Q'$.

3. a) Si $R \stackrel{x(w)}{\longrightarrow} R'$ y $Q \stackrel{\bar{x}y}{\longrightarrow} Q'$, por $R \rhd P$ y la Def. 2.46.3, se tiene que $\exists P'$ tal que $P \Longrightarrow P'' \stackrel{x(w)}{\longrightarrow} P'$ y $\forall z \; R'\{z/w\} \rhd P'\{z/w\}$, donde posiblemente P'' sea P. Entonces, por la Def. 2.23.2 se tiene que $P'' \diamond Q$. Por la Def. 2.23.3, y $Q \stackrel{\bar{x}y}{\longrightarrow} Q'$ se tiene que

$P'\{y/w\} \diamond Q'$. En particular, para $z = y$ se tiene que $R'\{y/w\} \rhd P'\{y/w\}$. Por tanto, $R'\{y/w\} \diamond_{her} Q'$.

b) Si $Q \xrightarrow{x(w)} Q'$ y $R \xrightarrow{\bar{x}y} R'$, por $R \rhd P$ y la Def. 2.46.2 se tiene que $P \Longrightarrow P'' \xrightarrow{\bar{x}y} P'$ donde $R' \rhd P'$ y posiblemente P'' sea P'. Entonces, por $P \diamond Q$ y las Def. 2.23.2 y 2.23.3 se tiene que $P'' \diamond Q$ y también que $P' \diamond Q'\{y/w\}$. Por tanto, $R' \diamond_{her} Q'\{y/w\}$.

4. a) Si $R \xrightarrow{x(w)} R'$ y $Q \xrightarrow{\bar{x}(w)} Q'$, por $R \rhd P$ y la Def. 2.46.3, se tiene que $\exists\, P'$ tal que $P \Longrightarrow P'' \xrightarrow{x(w)} P'$ y $\forall\, z\ R'\{z/w\} \rhd P'\{z/w\}$, donde posiblemente P'' sea P. Entonces, por la Def. 2.23.2 se tiene que $P'' \diamond Q$. Por la Def. 2.23.4, y $Q \xrightarrow{\bar{x}(w)} Q'$ se tiene que $P' \diamond Q'$. En particular, para $z = w$ se tiene que $R' \rhd P'$. Por tanto, $R' \diamond_{her} Q'$.

b) Si $Q \xrightarrow{x(w)} Q'$ y $R \xrightarrow{\bar{x}(w)} R'$, por $R \rhd P$ y la Def. 2.46.4 se tiene que $P \Longrightarrow P'' \xrightarrow{\bar{x}(w)} P'$ donde $R' \rhd P'$ y posiblemente P'' sea P'. Entonces, por $P \diamond Q$ y las Def. 2.23.2 y 2.23.4 se tiene que $P'' \diamond Q$ y también que $P' \diamond Q'$. Por tanto, $R' \diamond_{her} Q'$.

■

El lema que figura a continuación sirve de apoyo para la Demostración 2.52:

Lema 2.53 *Sea $R \rhd P$, donde P no presenta una traza infinita de transiciones τ, y $P \xrightarrow{\tau}$. Entonces, $\exists\, P', R'$ tales que $P \Longrightarrow P', R \Longrightarrow R', R' \rhd P'$ y $P' \not\xrightarrow{\tau}$.*

Demostración. Puede derivarse de la Def. 2.47. Bajo las condiciones del lema, se tiene que $\exists\, P_1, R_1$ tales que $P \xrightarrow{\tau} P_1, R \Longrightarrow R_1$ y $R_1 \rhd P_1$. Entonces, mientras $P_i \xrightarrow{\tau}$ aplicamos de nuevo la definición, obteniendo P_{i+1}, R_{i+1}, donde $R_{i+1} \rhd P_{i+1}$. Dado que P no presenta una traza infinita de transiciones τ, $\exists\, P_n, R_n$ tales que $R_n \rhd P_n$ y $P_n \not\xrightarrow{\tau}$. ■

Por tanto, la herencia mantiene la compatibilidad, y una única prueba de herencia asegura que un rol vástago puede sustituir a cualquiera de sus padres en cualquier contexto, sin necesidad de comprobar nuevamente la compatibilidad en la conexión afectada por la sustitución. Este resultado define cuándo un determinado componente ya existente puede ser utilizado en una arquitectura: los roles del componente deben heredar a aquéllos especificados en la arquitectura. Sirva el siguiente ejemplo para ilustrar estas afirmaciones.

Ejemplo 2.54 (Buffer) *Supongamos un componente* Buffer *que proporcione las operaciones* insertar *y* extraer *habituales. La especificación completa de este componente podría ser la siguiente:*

$$Buffer(insertar, extraer) = BufferSpec(insertar, extraer, \textsc{Null}, \textsc{True})$$

$$\begin{aligned}
&BufferSpec(insertar, extraer, nodo, vacío) = \\
&\quad insertar(ítem).(nuevo)(\ Nodo(nuevo, ítem, nodo, vacío)\ \mid \\
&\qquad\qquad BufferSpec(insertar, extraer, nuevo, \textsc{False})\ ) \\
&+\ extraer(return, error).(\ [vacío{=}\textsc{False}]\ nodo(ítem\ sig\ último).\overline{return}\ ítem. \\
&\qquad\qquad BufferSpec(insertar, extraer, sig, último) \\
&\qquad + \quad [vacío{=}\textsc{True}]\ \overline{error}.BufferSpec(insertar, extraer, sig, vacío)\ )
\end{aligned}$$

$$Nodo(nodo, ítem, siguiente, último) = \overline{nodo}\ ítem\ siguiente\ último.\mathbf{0}$$

Como vemos, la especificación tiene en cuenta la posibilidad de que la operación de extracción produzca un error si el Buffer *está vacío. Podemos descomponer la interfaz de este* Buffer *en dos roles,* Inserción *y* Extracción*, con la siguiente especificación:*

$$Inserción(insertar) = insertar(ítem).Inserción(insertar)$$

$$Extracción(extraer) = extraer(return, error).(\quad \tau.\overline{return}(ítem).Extracción(extraer) \\ + \quad \tau.\overline{error}.Extracción(extraer) \ )$$

De nuevo podemos apreciar cómo los roles proporcionan únicamente una especificación de la interfaz del componente: tras aceptar una operación de extracción, el Buffer *responderá a la misma enviando un* ítem*, en caso de que esté disponible, o con un* error *en caso de que esté vacío. Dado que la existencia o no de elementos en el* Buffer *es una condición interna que no puede ser determinada por quien observa el* Buffer *a través del enlace* extraer*, estas opciones figuran en el rol* Extracción *como parte de una decisión local del componente* Buffer*.*

Supongamos ahora que queremos utilizar nuestro componente Buffer *en un sistema Productor/Consumidor, donde actuará como mediador entre dos componentes, uno que actuará de* Productor *y otro que hará de* Consumidor*, de acuerdo respectivamente a los roles:*

$$Productor(insertar) = \overline{insertar}(ítem).Productor(insertar)$$

$$Consumidor(extraer) = \overline{extraer}(return).return(ítem).Consumidor(extraer)$$

Para determinar si la arquitectura de nuestro sistema es segura deberemos comprobar, haciendo uso de la Definición 2.23, la compatibilidad de las conexiones {Productor*,* Inserción} *y* {Consumidor*,* Extracción}*. Como resulta evidente, en este último caso no podremos determinar la compatibilidad, puesto que el rol* Consumidor *descrito no tiene en cuenta la posibilidad de que el* Buffer *esté vacío.*

No obstante, podemos escribir una variante más detallada del rol Consumidor *que gestione la excepción correspondiente a la situación de buffer vacío:*

$$Consumidor'(extraer) = \overline{extraer}(return\ error). \\ (return(ítem).Consumidor'(extraer) + error.Consumidor'(extraer) \ )$$

Tenemos entonces que se dan las siguientes situaciones de compatibilidad:

$$Productor(insertar) \diamond \ Inserción(insertar)$$
$$Consumidor(extraer) \not\diamond \ Extracción(extraer)$$
$$Consumidor'(extraer) \diamond \ Extracción(extraer)$$

Llegados a este punto, podemos construir una variante de nuestro Buffer *que presente un comportamiento más amistoso, de forma que, en caso de estar vacío, devuelva un ítem con un valor especial,* Nulo*, en vez de producir un error. La especificación (incompleta) del nuevo componente, junto con su rol* Extracción′ *que lo representa es la siguiente:*

$$\begin{aligned}
&\mathit{BufferSpec}(\mathit{insertar}, \mathit{extraer}, \mathit{nodo}, \mathit{vacío}) = \ldots \\
&\quad + \ \mathit{extraer}(\mathit{return}, \mathit{error}). \\
&\qquad (\ [\mathit{vacío}{=}\textsc{False}] \ \mathit{nodo}(\mathit{ítem}, \mathit{siguiente}, \mathit{último}).\overline{\mathit{return}} \ \mathit{ítem}. \\
&\qquad\quad \mathit{BufferSpec}(\mathit{insertar}, \mathit{extraer}, \mathit{siguiente}, \mathit{último}) \\
&\qquad + \ \ [\mathit{vacío}{=}\textsc{True}] \ \overline{\mathit{return}} \ \textsc{Nulo}. \\
&\qquad\quad \mathit{BufferSpec}(\mathit{insertar}, \mathit{extraer}, \mathit{siguiente}, \mathit{vacío}) \ )
\end{aligned}$$

$$\mathit{Extracción}'(\mathit{extraer}) = \mathit{extraer}(\mathit{return})\overline{\mathit{return}}(\mathit{ítem}).\mathit{Extracción}'(\mathit{extraer})$$

Se observa ahora que, independientemente de la situación del Buffer*, el consumidor recibirá un* ítem *en respuesta a su petición de extracción (aunque ese ítem pueda ser nulo). Por medio de la Definición 2.23 podemos comprobar que ahora sí se tiene que:*

$$\mathit{Extracción}'(\mathit{extraer}) \diamond \mathit{Consumidor}(\mathit{extraer})$$

rol este último con el cual Extracción *no era compatible. En este sentido, podríamos decir que el rol* Extracción′ *relaja las exigencias del rol* Extracción*. Llegados a este punto, mediante la relación de herencia de comportamiento de la Definición 2.47 podremos comprobar que:*

$$\mathit{Extracción}'(\mathit{extraer}) \rhd \mathit{Extracción}(\mathit{extraer})$$

es decir, que existe una relación de herencia entre ambos roles, debido a que el primero ha suprimido las decisiones locales del segundo. Por tanto, y de acuerdo con el resultado del Teorema 2.52, tendremos que el rol vástago es compatible con aquellos roles que sean compatibles con su progenitor, y en particular que:

$$\mathit{Extracción}'(\mathit{extraer}) \diamond \mathit{Consumidor}'(\mathit{extraer})$$

La Figura 2.13 muestra todas las relaciones entre los roles descritos en este ejemplo. □

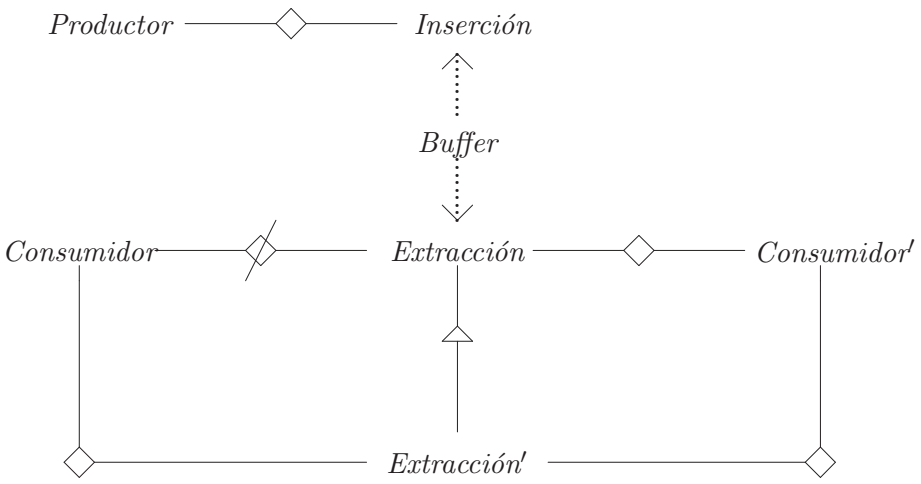


Figura 2.13: Relaciones entre los roles del Ejemplo 2.54.

Aunque el ejemplo que acabamos de presentar es muy simple, sirve para mostrar las conexiones entre compatibilidad y herencia, y cómo esta última permite modelar la relación entre distintas versiones del mismo componente de software, lo que puede ser utilizado como soporte para la especificación incremental.

2.4.2 Relación de extensión

La relación de herencia presentada en la sección precedente es demasiado restrictiva. La Definición 2.43 establece que el rol vástago no puede extender a sus padres añadiendo nuevo comportamiento o funcionalidad. Podemos superar estas restricciones definiendo la *extensión* como sigue:

Definición 2.55 (Extensión de comportamiento) R extiende *a* P*, lo que se representa como* $R \rhd\!\!\rhd P$*, si y sólo si*

$$(fn(R) - fn(P))R \rhd P$$

Esta definición relaciona extensión y herencia, y establece las condiciones que aseguran que el rol extendido R pueda ser conectado con éxito con cualquier $Q \in P\diamondsuit$. Con objeto de mantener la compatibilidad, se restringe el comportamiento adicional en el agente vástago R, asegurando que, si R y Q se conectan, R se comportará tal como lo hacía P (con la diferencia de que algunas decisiones locales de P pueden haber sido redefinidas u omitidas por R). La funcionalidad adicional que proporciona el agente vástago R no va a ser utilizada en su conexión con Q. No obstante, este comportamiento adicional de R, puede ser usado con éxito en otros contextos o arquitecturas, incluso en combinación con Q.

Ejemplo 2.56 (Protocolo de charla) *Juan, Pablo, y María son amigos y acostumbran a reunirse por parejas para charlar durante un rato. Cuando dos de ellos se encuentran, se saludan uno al otro diciendo 'hola', y acuerdan hablar de un cierto 'tema'. Al separarse, se dicen 'adiós'. Juan únicamente es capaz de hablar sobre un tema en cada conversación. Por el contrario, María puede acordar con su compañero el cambiar de tema durante el transcurso de la conversación. Por último, Pablo parece aceptar el que se cambie de tema, pero sigue hablando de lo mismo, obstaculizando la conversación. El comportamiento de estos tres personajes es el especificado por los siguientes roles:*

$$\begin{aligned}
&Juan(hola, adiós) = \overline{hola}(tema).JuanCharlando(hola, tema, adiós)\\
&JuanCharlando(hola, tema, adiós) = \tau.\overline{tema}.JuanCharlando(hola, tema, adiós)\\
&\qquad\qquad + \tau.\overline{adiós}.Juan(hola, adiós)
\end{aligned}$$

$$\begin{aligned}
&Pablo(hola, cambio, adiós) = hola(tema).PabloCharlando(hola, tema, cambio, adiós)\\
&PabloCharlando(hola, tema, cambio, adiós) =\\
&\quad tema.PabloCharlando(hola, tema, cambio, adiós)\\
&+\ cambio(nuevotema).PabloCharlando(hola, tema, cambio, adiós)\\
&+\ adiós.Pablo(hola, cambio, adiós)
\end{aligned}$$

$$\begin{aligned}
&María(hola, cambio, adiós) = \overline{hola}(tema).MaríaCharlando(hola, tema, cambio, adiós)\\
&MaríaCharlando(hola, tema, cambio, adiós) =\\
&\quad \tau.\overline{tema}.MaríaCharlando(hola, tema, cambio, adiós)\\
&+\ \overline{cambio}(nuevotema).MaríaCharlando(hola, nuevotema, cambio, adiós)\\
&+\ \tau.\overline{adiós}.María(hola, cambio, adiós)
\end{aligned}$$

Obsérvese primero que Juan $\diamond$ Pablo, dado que el primero nunca intentará cambiar de conversación y por lo tanto no notará el comportamiento impropio de éste último. Sin embargo, María sí se dará cuenta de ello, por lo que no es compatible con Pablo. Obsérvese también que María $\not\triangleright$ Juan, sino que María $\rhd\!\!\rhd$ Juan. Por tanto, podemos inferir que (cambio)María $\diamond$ Pablo, es decir, suponiendo que María no intente cambiar de tema de conversación, su comportamiento será compatible con el de Pablo. La Figura 2.14 muestra las relaciones entre los roles de este ejemplo.

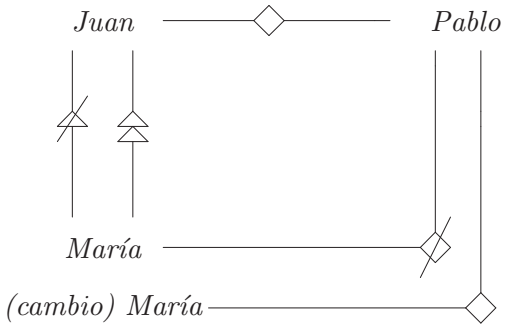


Figura 2.14: Relaciones entre los roles del Ejemplo 2.56

Este ejemplo toma un cariz distinto si consideramos a Juan como un servidor orientado a la conexión (como un servidor TCP, por ejemplo), en el sentido de que usa un único canal (tema) para cada sesión o 'conversación' con un cliente. Por el contrario, María actúa como un servidor orientado a la transacción, debido a que puede cambiar el canal de comunicación durante la transmisión (como un servidor UDP). Por último, podemos considerar a Pablo como un cliente malintencionado. □

Por tanto, si $R \rhd\!\!\rhd P$ y $P \diamond Q$, la relación de extensión asegura el reemplazamiento de P por R, pero sin cambiar las características de la arquitectura en la que se produce este reemplazamiento. Sin embargo, también es posible que $R \diamond Q$ sin restringir el comportamiento adicional. En este caso, el reemplazamiento de P por R implica un cambio en las características de la arquitectura, y el comportamiento adicional de R es utilizado en el sistema resultante. En este caso, la arquitectura en la que se produce la conexión de P y Q describe una familia de productos de software similares pero no idénticos. Bajo ciertas condiciones, es posible obtener un resultado similar al del Teorema 2.52 para esta familia de sistemas con un patrón arquitectónico común, en el que vamos añadiendo comportamiento a los roles a medida que los derivamos por extensión unos de otros:

Teorema 2.57 *Sea $P \diamond Q$. Sea $R = P + P'$. Si $P' \diamond Q$ se tiene que $R \diamond Q$.*

Demostración. (Obsérvese que ésta también es una demostración para el Teorema 2.30.d.) El resultado se deriva de las Definiciones 2.23 y 2.47. Para demostrarlo, basta con probar que $\diamond_{ext} = \{(R, Q) \,:\, \exists\, P, P' \,.\, R = P + P' \wedge P \diamond Q \wedge P' \diamond Q\}$ es una relación de compatibilidad.

Supongamos que $R \diamond_{ext} Q$, entonces tenemos que comprobar las condiciones de la Definición 2.23.

1. a) Si R no tiene éxito, entonces bien P o P' (o ambos) no tienen éxito. Según esto, de $P \diamond Q$, $P' \diamond Q$ y la Def. 2.23.1 se tiene que Q es sincronizable tanto con P como con P'. Por tanto, se tiene que $R = P + P'$ es sincronizable con Q.

   b) Por otro lado, si Q no tiene éxito, por $P \diamond Q$, $P' \diamond Q$ y la Def. 2.23.1 se tiene que Q es sincronizable tanto con P como con P'. Por tanto, $\exists\, \alpha, \beta$ tales que $P \stackrel{\alpha}{\Longrightarrow}$, $P' \stackrel{\beta}{\Longrightarrow}$, $Q \stackrel{\overline{\alpha}}{\Longrightarrow}$ $Q \stackrel{\overline{\beta}}{\Longrightarrow}$ (donde $\overline{\alpha}$, $\overline{\beta}$ representan acciones complementarias a α y β, respectivamente). De ahí que, incluso en presencia de transiciones τ antes de α o β se tiene $R = P + P'$ es sincronizable con Q.

2. a) Si $R \stackrel{\tau}{\longrightarrow} R'$, entonces bien $P \stackrel{\tau}{\longrightarrow} R'$ o bien $P' \stackrel{\tau}{\longrightarrow} R'$. En cualquier caso, de $P \diamond Q$, $P' \diamond Q$ y la Def. 2.23.2 se tiene que $R' \diamond Q$. Si consideramos $R'' = R' + R'$ tendremos que $R'' \diamond_{ext} Q$, donde obviamente $R'' \equiv R'$.

   b) Si $Q \stackrel{\tau}{\longrightarrow} Q'$, por $P \diamond Q$, $P' \diamond Q$ y la Def. 2.23.2 se tiene que $P \diamond Q'$ y también que $P' \diamond Q'$. Dado que $R = P + P'$, se infiere que $R \diamond_{ext} Q'$.

3. a) Si $R \stackrel{x(w)}{\longrightarrow} R'$ y $Q \stackrel{\bar{x}y}{\longrightarrow} Q'$, entonces $P \stackrel{x(w)}{\longrightarrow} R'$ o bien $P' \stackrel{x(w)}{\longrightarrow} R'$ (o ambos). En cualquier caso, por $P \diamond Q$, $P' \diamond Q$ y la Def. 2.23.3 se tiene que $R'\{y/w\} \diamond Q'$. Si consideramos $R'' = R' + R'$ tendremos que $R''\{y/w\} \diamond_{ext} Q'$, donde obviamente $R'' \equiv R'$.

   b) De forma similar, si $Q \stackrel{x(w)}{\longrightarrow} Q'$ y $R \stackrel{\bar{x}y}{\longrightarrow} R'$, entonces bien P o P' (o ambos) presentarán una transición $\bar{x}y$ a R'. Entonces, por $P \diamond Q$, $P' \diamond Q$ y la Def. 2.23.3 se tiene que $R' \diamond Q'\{y/w\}$. De nuevo, si consideramos $R'' = R' + R'$ se tendrá que $R'' \diamond_{ext} Q'\{y/w\}$, donde obviamente $R'' \equiv R'$.

4. a) Si $R \stackrel{x(w)}{\longrightarrow} R'$ y $Q \stackrel{\bar{x}(w)}{\longrightarrow} Q'$, entonces bien $P \stackrel{x(w)}{\longrightarrow} R'$ o bien $P' \stackrel{x(w)}{\longrightarrow} R'$ (o ambos). En cualquier caso, por $P \diamond Q$, $P' \diamond Q$ y la Def. 2.23.4 se tiene que $R' \diamond Q'$. Si consideramos $R'' = R' + R'$ se tendrá que $R'' \diamond_{ext} Q'$, donde obviamente $R'' \equiv R'$.

   b) De forma similar, si $Q \stackrel{x(w)}{\longrightarrow} Q'$ y $R \stackrel{\bar{x}(w)}{\longrightarrow} R'$, entonces bien P o P' (o ambos) presentarán una transición $\bar{x}(w)$ a R'. Entonces, por $P \diamond Q$, $P' \diamond Q$ y la Def. 2.23.4 se tiene que $R' \diamond Q'$. De nuevo, si consideramos $R'' = R' + R'$ se tendrá que $R'' \diamond_{ext} Q'$, donde obviamente $R'' \equiv R'$. ■

Por tanto, una vez que hemos demostrado que una conexión entre dos roles P y Q es compatible, para cualquier rol $R = P + P'$ que extienda P, sólo necesitamos comprobar la compatibilidad del comportamiento adicional P'. Este resultado justifica la introducción de construcciones lingüísticas relacionadas con la herencia en un lenguaje de descripción de arquitectura de alto nivel, con objeto de ocultar a los ingenieros del software la complejidad de la relación de herencia. Esto es lo que se hará en el capítulo siguiente cuando presentemos el lenguaje LEDA.

2.5 Trabajos relacionados

En los últimos años la Arquitectura del Software ha venido recibiendo un activo interés por parte de la comunidad científica [Garlan y Perry, 1995], y se ha ido sucediendo toda una serie de propuestas para la descripción, análisis y desarrollo de la arquitectura de los sistemas de software. Muchas de estas propuestas están fundamentadas en algún formalismo, lo que permite realizar algún tipo de verificación de las arquitecturas descritas, bien para comprobar si se

ajustan a las restricciones de un determinado estilo arquitectónico o para verificar si se cumplen diversas propiedades. Así, varias álgebras de procesos, tales como CSP [Allen y Garlan, 1997], el cálculo π [Magee y Kramer, 1996], o ACP [Bergstra y Klint, 1998], y también otros formalismos, como la Máquina Química Abstracta (CHAM) [Inverardi y Wolf, 1995], la lógica temporal de acciones (TLA) y los lenguajes de coordinación [Ciancarini et al., 1998], la notación Z [Abowd et al., 1993], los conjuntos de eventos parcialmente ordenados [Luckham et al., 1995, Vera et al., 1999], o las gramáticas de grafos [LeMétayer, 1998, Hirsch et al., 1999], han sido propuestas como base formal de notaciones, herramientas y entornos de descripción de arquitectura. En esta sección comentaremos los puntos en común y diferencias entre nuestra propuesta y algunas de las más relevantes que pueden encontrarse en la literatura al respecto, en particular con aquéllas que son más próximas a nuestro enfoque.

En primer lugar, debemos decir que la motivación básica que está detrás de todos estos trabajos es la de identificar los bloques principales —o componentes— que participan en la construcción de sistemas de software. No obstante, algunas propuestas [Shaw et al., 1995, Allen y Garlan, 1994] diferencian entre *componentes*, descritos por medio de un conjunto de *puertos*, y *conectores*, utilizados como nexo de unión entre componentes, y que son descritos por medio de un conjunto de *roles*.

Desde nuestro punto de vista, y tal como hemos indicado en la Sección 1.6, la distinción entre componentes y conectores es, por lo general, demasiado sutil, por lo que nuestra propuesta no distingue entre estas dos categorías, sino que ambas —componentes y conectores— son considerados genéricamente como componentes, siendo todos ellos descritos por medio de un conjunto de roles que representan su interfaz. Esta decisión simplifica además la formalización de las definiciones de rol, compatibilidad y herencia, y los resultados correspondientes relativos a la composición con éxito de roles, componentes y arquitectura.

La utilidad de realizar de forma separada la especificación de interacciones no relacionadas entre sí, aunque correspondan al mismo componente ha sido postulada también por otros autores, como [Honda et al., 1998]. En este trabajo se denomina *sesión* a las interacciones que se establecen entre los componentes de un sistema, tomados dos a dos, y se propone el uso de un lenguaje basado en el cálculo π para su especificación. Este concepto de sesión coincide básicamente con lo que nosotros definimos como roles, aunque en nuestro trabajo las conexiones no se limitan únicamente a relacionar dos roles, como ya hemos visto, lo que permite describir —y por tanto analizar— interacciones más complejas. También en este sentido, una propuesta reciente, debida a Brogi y colaboradores [Bracciali et al., 2001], postula la descripción de patrones de interacción finitos utilizando para ello un subconjunto del cálculo π en el que se prescinde de la recursividad en la definición de agentes con objeto de reducir el coste computacional de las comprobaciones de compatibilidad o reemplazabilidad. Si bien esta restricción no garantiza la corrección de los sistemas analizados, sus autores arguyen que permite eliminar de forma sencilla un gran número de errores de diseño.

Siguiendo con otras propuestas, algo más alejadas de nuestro contexto, la arquitectura ToolBus [Bergstra y Klint, 1998] utiliza especificaciones de interfaz, denominadas *scripts* o guiones, escritas en una extensión temporal del álgebra de procesos ACP, lo que permite la especificación de restricciones temporales de las arquitecturas. No obstante, el propósito de estos guiones es fundamentalmente descriptivo, no analítico. Por otro lado, esta propuesta se restringe a la especificación de sistemas siguiendo una arquitectura concreta, mientras que nuestro objetivo es abordar la descripción de arquitecturas de software arbitrarias.

Como ya hemos indicado en la Sección 1.4.5, Rapide [Luckham et al., 1995] es un ADL basado en eventos en el que el comportamiento de los componentes queda reflejado mediante patrones de eventos que describen conjuntos de eventos parcialmente ordenados (*posets*). En esta propuesta, la propiedad clave a comprobar es la identificación y ordenación de eventos, considerados como hitos abstractos que representan el progreso computacional del sistema. Los eventos pueden describirse con diversos grados de detalle, según interese, lo que dota de cierta flexibilidad a esta propuesta. Utilizando Rapide, la consistencia entre las interfaces descritas de esta forma se comprueba mediante simulación. Cada simulación genera un *poset* que representa una determinada interacción entre los componentes. Las ordenaciones correctas o admisibles se describen imponiendo restricciones sobre los *posets*.

En [Magee et al., 1995], el cálculo π es utilizado para definir la semántica del ADL Darwin [Magee y Kramer, 1996]. Aunque inicialmente la comprobación de tipos se reducía en Darwin a verificar la equivalencia de nombres, y no se describía el comportamiento de las conexiones entre componentes, en trabajos posteriores [Magee et al., 1999] se propone la especificación del comportamiento de los componentes por medio de sistemas de transiciones etiquetadas (LTS) de estado finito. A continuación, estos LTS son analizados utilizando el entorno TRACTA [Giannakopoulou et al., 1999]. Como en nuestra propuesta, la descripción de interfaces por medio de roles y la composición jerárquica son utilizados para minimizar la explosión del número de estados. Sin embargo, el formalismo utilizado en estos últimos trabajos ya no es el cálculo π, y carece de la posibilidad de expresar movilidad, lo que no le hace adecuado para la descripción de arquitecturas dinámicas. En estos trabajos tampoco se consideran aspectos como la herencia o el refinamiento de comportamientos.

Nuestra definición de compatibilidad sigue las ideas desarrolladas en [Allen y Garlan, 1997], donde se propone CSP para la especificación de componentes en el ADL Wright. No obstante, y como ya hemos destacado anteriormente, formalismos como CSP o CCS no son adecuados para la descripción de estructuras que presenten una topología de comunicación cambiante. Como máximo, CSP puede utilizarse para la descripción de sistemas con un número limitado de configuraciones, como los propios autores muestran en [Allen et al., 1998], pero no en sistemas fuertemente dinámicos, donde formalismos como el cálculo π se muestran mucho más adecuados. Por este motivo, nuestra definición de compatibilidad tiene en cuenta la movilidad, a diferencia de la propuesta que acabamos de citar.

El trabajo de Allen y Garlan se basa en el concepto de refinamiento en CSP. En su artículo, el refinamiento (que podríamos describir a grandes rasgos como una relación de inclusión entre procesos) se utiliza para definir una relación de compatibilidad entre dos categorías asimétricas de interfaces: los puertos (que representan componentes), y los roles (que representan conectores entre componentes). Un puerto es compatible con un rol si el primero refina al segundo. Como ya hemos indicado, nuestro enfoque difiere del de estos autores, en que no distinguimos entre componentes y conectores, con lo que la representación de las interfaces de estos elementos se realiza siempre de la misma manera, y las conexiones se llevan a cabo entre roles que presentan acciones complementarias —de distinto signo—, y representan a componentes que realizan también acciones complementarias.

Otra diferencia con [Allen y Garlan, 1997] es que nuestra propuesta implica una metodología para la especificación de los roles de un componente, y no requiere de una transformación de las interfaces (como su versión determinista de los roles) para realizar la comprobación de compatibilidad. Por otra parte, en este trabajo no se abordan aspectos de herencia ni extensión, que en nuestra propuesta se usan para definir las condiciones de reemplazamiento de un componente asegurando la preservación de la compatibilidad de la arquitectura. Aunque la noción de refinamiento en CSP puede posiblemente utilizarse para definir una relación de

herencia similar a la propuesta en este trabajo, se trata éste de un aspecto no abordado en los trabajos de Allen y Garlan, en los que las referencias a refinamiento se hacen siempre en el contexto de la compatibilidad entre procesos.

Esta misma noción de compatibilidad está presente también en otros trabajos relevantes, tales como [Compare et al., 1999] o [Yellin y Strom, 1997]. En el primero de ellos se propone la CHAM para la especificación de la arquitectura del software, y se utilizan dos tipos diferentes de análisis para la verificación de propiedades de viveza de una arquitectura. Sin embargo, tampoco aquí se abordan aspectos de herencia o extensión. Por otro lado, en [Yellin y Strom, 1997] se utilizan diagramas de estado finito para la especificación de lo que los autores denominan *protocolos*, y se presentan relaciones de compatibilidad y *subtipado* de protocolos. Aunque nuestro trabajo llega a resultados similares a los contenidos en el citado trabajo, conseguimos superar algunas de sus limitaciones. En primer lugar, nuestra propuesta aborda la especificación y análisis de sistemas dinámicos, mientras que la suya puede aplicarse únicamente a componentes estáticos. En segundo lugar, nuestro trabajo utiliza la restricción del ámbito de los nombres de enlace para obtener especificaciones modulares de componentes y roles, mientras que en [Yellin y Strom, 1997] todos los mensajes se envían a través de un espacio de nombres común, lo que lleva con facilidad a cometer errores durante la especificación. En tercer lugar, nuestra propuesta utiliza decisiones locales y globales para determinar las responsabilidades de acción y reacción, mientras que la suya sólo tiene en cuenta decisiones globales síncronas. Por último, en la propuesta citada, si un protocolo presenta una acción de salida, cualquier protocolo que sea compatible con éste deberá presentar la acción de entrada complementaria, de manera que no pueda ocurrir lo que los autores denominan como *recepciones indefinidas*. Por el contrario, con nuestra propuesta se puede comprobar la compatibilidad entre agentes con conjuntos de nombres distintos —y que realicen, por tanto, conjuntos de acciones también distintos—, lo que permite la combinación de componentes que encajan sólo de manera parcial.

Respecto a la herencia o *subtipado* de comportamiento, este concepto fue desarrollado inicialmente en [America, 1991], y recogido posteriormente en diversos trabajos [Liskov y Wing, 1994, Dhara y Leavens, 1996, Zaremski y Wing, 1997, Leavens y Dhara, 2000]. Esta idea está ligada al concepto de polimorfismo en el contexto de los lenguajes orientados a objetos, donde la herencia aparece como condición natural para el reemplazamiento fiable, lo que ha sido estudiado en diversos trabajos, en particular en [Mikhajlov y Sekerinski, 1998] y [Mikhajlova, 1999]. En el primero de estos trabajos, estos aspectos se estudian en un contexto diferente al nuestro: el del problema de la *clase base frágil*, llegando a formalizar las condiciones que debe cumplir la relación de herencia entre clases para evitar la aparición de este problema. En el segundo, más próximo a nuestro enfoque, se abordan las condiciones necesarias para asegurar el reemplazamiento polimórfico de un componente de un sistema por una versión más concreta y especializada del mismo. El formalismo utilizado en los citados trabajos es el cálculo de refinamientos [Back y von Wright, 1998], que permite para razonar sobre la corrección y refinamiento de programas imperativos. En ambos trabajos, el problema se aborda desde el punto de vista del refinamiento de clases, interfaces y signaturas de métodos, pero no en el de las interacciones entre componentes, como es nuestro caso.

La noción de herencia y extensión de comportamiento ofrecida en este trabajo presenta ciertas analogías con la noción de *refinamiento de acciones* en el contexto de álgebras de procesos [Aceto y Hennessy, 1994, Gorrieri y Rensink, 2000]. La motivación subyacente en ambas propuestas es similar, y ambas exploran las relaciones entre sucesivas versiones de componentes y especificaciones. Sin embargo, el refinamiento de acciones se refiere a una relación entre agentes por medio de la cual una acción atómica en un agente se reemplaza mediante un proceso completo en el otro. Ambos agentes describen, por tanto, *el mismo* componente, pero con diferentes

grados de abstracción, con lo que el refinamiento de acciones puede utilizarse como guía en el paso de la especificación a la implementación, tal como se muestra en [Gorrieri y Rensik, 1997], al añadir detalles adicionales a los componentes a lo largo del proceso de desarrollo. Por el contrario, nuestra relación de herencia permite el reemplazamiento de un componente por *otro diferente*, por lo general de similar nivel de abstracción. Este segundo componente ha de presentar unos roles que exijan requisitos más débiles o ofrezcan una funcionalidad más amplia que el original, de forma análoga a como se utiliza la herencia en el paradigma orientado a objetos (lo que motiva precisamente el nombre de nuestra relación). Por tanto, ambas propuestas son complementarias.

La relación de herencia que hemos presentado en este capítulo es en cierto modo *restrictiva*, en el sentido de que se exigen numerosas condiciones para que podamos considerar que un rol hereda de otro. Esto es necesario debido a que hemos definido la herencia de forma que asegure la reemplazabilidad de un componente por otro *en cualquier contexto*. No obstante, parece razonable suponer que algunas de estas condiciones pueden relajarse si el objetivo se reduce a asegurar la reemplazabilidad en un contexto determinado —y así la relación de extensión que hemos presentado hace referencia a las propiedades de la herencia en el contexto de una arquitectura determinada. En este mismo sentido, se han propuesto relaciones dependientes del contexto [Larsen, 1987], aunque referidas a la bisimulación y no a una relación más general de derivación o herencia. Estas relaciones definen cuándo dos procesos pueden ser considerados equivalentes desde el punto de vista del contexto o entorno en el que los situemos. Dicho entorno es modelado mediante un proceso, y puede por tanto ir evolucionando.

2.6 Conclusiones

En este capítulo hemos mostrado cómo el cálculo π puede ser utilizado como soporte formal para la especificación de la arquitectura del software, y hemos presentado definiciones que formalizan las nociones de rol, conexión y arquitectura en el contexto de este cálculo.

A continuación, hemos mostrado cómo es posible verificar formalmente ciertas propiedades de interés (como el éxito en la conexión de dos o más componentes), para lo que hemos definido una relación de compatibilidad entre agentes en el cálculo π. También hemos demostrado que la compatibilidad implica la conexión con éxito.

Por último, hemos presentado una relación de herencia entre componentes que permite el refinamiento de los mismos, junto con la adición de nuevos comportamientos. Esta relación de herencia asegura además la reemplazabilidad, al preservar la compatibilidad del componente derivado con todos aquellos componentes que fuesen compatibles con sus progenitores.

Mediante la utilización de un sistema de transiciones especializado, como el propuesto por Sangiorgi para la bisimulación, es posible desarrollar herramientas que comprueben de forma automática las relaciones que hemos definido en este capítulo. Actualmente, y a partir del MWB, podemos analizar la compatibilidad entre roles.

Todos estos resultados son de utilidad dentro del campo de la especificación formal de la arquitectura del software. Sin embargo, no debemos perder de vista que el cálculo π es una notación de bajo nivel, lo que hace farragosa su utilización práctica. Por este motivo se impone el desarrollo de un lenguaje de descripción de arquitectura, de nivel más alto, que incorpore los conceptos a que se hace referencia en este capítulo. De la descripción de dicho lenguaje arquitectónico es de lo que trata el capítulo siguiente.

Capítulo 3

El lenguaje de descripción de arquitectura

El interés de representar de forma explícita la arquitectura de los sistema de software es evidente, puesto que es en este nivel arquitectónico donde se aborda de forma natural todo lo referente a las propiedades estructurales de estos sistemas. Estas representaciones permiten aumentar el grado de abstracción, facilitando la descripción y comprensión de sistemas complejos. Además, favorecen la reutilización tanto de la arquitectura en sí, como de los componentes que la constituyen [Shaw y Garlan, 1996].

Desde el punto de vista de la Arquitectura del Software, los sistemas de software consisten en una colección de elementos computacionales y de datos, o *componentes*, interconectados de una forma determinada. Por tanto, a la hora de describir la estructura de un sistema, indicaremos cuáles son sus componentes. Sin embargo, no sólo los componentes, sino también las arquitecturas, los mecanismos de interacción y los patrones arquitectónicos deben ser conceptos muy a tener en cuenta en cualquier lenguaje de descripción de arquitectura (ADL).

A pesar de lo reciente de este campo, hasta la fecha se han propuesto toda una serie de ADLs. Una visión general de estos lenguajes y sus principales características puede adquirirse a través de los trabajos de Medvidovic [Medvidovic y Rosenblum, 1997, Medvidovic y Taylor, 2000]. Los ADLs tratan de dar solución a la necesidad de contar con notaciones dotadas de la suficiente expresividad a la hora de realizar el diseño arquitectónico de una aplicación de software. Estos lenguajes permiten realizar descripciones precisas de cómo se combinan los componentes para formar sistemas mayores. Sin embargo, una parte importante de las notaciones propuestas carecen de una base formal adecuada. Así mismo, aspectos significativos tales como la descripción de sistemas cuya arquitectura sea dinámica, la parametrización de arquitecturas o su refinamiento, no son por lo general tenidos en cuenta.

Una arquitectura dinámica es aquélla en la que se crean, interconectan y eliminan componentes durante la ejecución del sistema, y que permite la reconfiguración, en tiempo de ejecución, de la topología de comunicación del mismo.

En el capítulo anterior hemos mostrado cómo el cálculo π puede ser utilizado para la descripción y análisis de arquitecturas de software. Sin embargo, es evidente que se trata de una notación de relativo bajo nivel, lo que hace difícil su aplicación como tal a la especificación de sistemas complejos. Ésta fue nuestra motivación original para el desarrollo de LEDA, un lenguaje formal de descripción de arquitectura que incorpora mecanismos de herencia y reconfiguración dinámica a cuya presentación está dedicado este capítulo.

3.1 El lenguaje LEDA

LEDA es un lenguaje de especificación para la descripción y validación de propiedades estructurales y de comportamiento de los sistemas de software. El lenguaje está articulado en dos niveles diferentes, uno de ellos para la definición de *componentes* y el otro para la definición de *roles*. Los componentes representan partes o módulos del sistema, y cada uno de ellos proporciona una determinada funcionalidad al mismo. Por su parte, los *roles* describen el comportamiento observable de los componentes, lo que será utilizado para el prototipado, validación y ejecución de la arquitectura.

Los roles se especifican por medio de una extensión del cálculo π, lo que permite la descripción de arquitecturas dinámicas. Cada rol proporciona una visión parcial de la interfaz de comportamiento de un componente, es decir, del protocolo que sigue en su interacción con el resto de los componentes del sistema.

Por otro lado, los componentes se describen como compuestos por otros componentes. La estructura o arquitectura de un componente se indica mediante las relaciones que se establecen entre sus subcomponentes, lo que se expresa mediante un conjunto de *conexiones* entre los roles de dichos subcomponentes.

Una de las características que diferencian a LEDA de otros ADLs es que no realiza distinción, en el nivel del lenguaje, entre componentes y conectores, es decir, los conectores se especifican como un tipo más de componentes. Esto permite que el lenguaje sea más simple y regular, a la vez que no impone un modelo composicional concreto para la descripción de arquitecturas de software.

Debido a que la semántica de LEDA está escrita en términos del cálculo π (véase la Sección 3.9), las especificaciones pueden ser tanto analizadas como ejecutadas, permitiendo esto último el prototipado de la arquitectura. A este respecto, y tal como se mostró en el Capítulo 2, es posible determinar si una arquitectura es composicional, es decir si sus componentes presentan comportamientos compatibles entre sí y pueden ser combinados con éxito para construir el sistema.

Por otra parte, el disponer de un mecanismo de adaptación de un componente a una interfaz que no sea compatible con la suya propia fomentaría la reutilización de componentes de software. Esto es para lo que sirven los *adaptadores* en el lenguaje LEDA. Estos adaptadores son pequeños elementos, similares a roles y también descritos utilizando el cálculo π, que son capaces de interconectar con éxito componentes cuyo comportamiento no es compatible.

Nuestro lenguaje se completa con mecanismos de herencia y parametrización de roles y componentes que aseguran la preservación de la compatibilidad. Mediante estos mecanismos, podemos reemplazar un componente en una arquitectura por otro que herede del primero, con la certeza de que esta sustitución no afectará a la composicionalidad del sistema.

Esto da lugar a un mecanismo de parametrización por el cual las arquitecturas descritas en LEDA pueden ser consideradas como marcos de trabajo genéricos que pueden ser instanciados y reutilizados tantas veces como sea necesario. Los marcos de trabajo composicionales se derivan del concepto de patrón arquitectónico, del que trata la Sección 1.2.2, y constituyen el máximo grado de reutilización en el desarrollo de software; no sólo los componentes, sino también el propio diseño arquitectónico es reutilizado en las aplicaciones construidas sobre el marco de trabajo [Pree, 1996]. En este sentido, las especificaciones en LEDA pueden considerarse como patrones o marcos de trabajo arquitectónicos genéricos que pueden ser extendidos y reutilizados, adaptándolos a nuevos requisitos.

3.2 Componentes y roles

Como ya hemos indicado, componentes y roles son los dos niveles en los que se articula el lenguaje LEDA. La especificación de estas dos categorías de elementos se hace de forma modular, lo que facilita la reutilización por separado tanto de unos como de otros.

LEDA distingue entre clases e instancias de componentes, y proporciona mecanismos para la extensión y derivación de componentes. La sintaxis completa utilizada para la especificación de una clase de componentes en LEDA se puede encontrar en el Apéndice A, pero podemos resumirla aquí como sigue:

```
component ClaseDeComponentes {
  constant
    listaDeConstantes;
  var
    variable : Tipo := valorInicial;
    ...
  interface
    instanciaDeRol : ClaseDeRoles;
    ...
  composition
    instanciaDeComponente : ClaseDeComponentes;
    ...
  attachments
    listaDeConexiones;
  }
```

Como podemos ver, después de la palabra reservada **component** viene el nombre de la clase de componentes que estamos describiendo, y a continuación, entre llaves, el cuerpo de la especificación. Éste se divide en cinco secciones principales: declaración de constantes, declaración de variables, descripción de la interfaz, composición y lista de conexiones.

Las declaraciones de constantes y variables figuran, respectivamente, a continuación de las palabras reservadas **constant** y **var**. El resto de elementos de la especificación podrá hacer referencia a estas constantes y variables, que servirán para coordinar y comunicar unos con otros estos elementos en aquellos casos en que sea necesario. En particular, los valores de las variables determinan el estado del componente. Para cada una de estas variables se puede indicar un valor inicial, aquél que toman cuando se crea el componente.

Por su parte, la descripción de la interfaz del componente viene indicada por la palabra reservada **interface**, seguida de una lista de instancias de roles, para cada uno de los cuales se indica su clase. Por tanto, la interfaz de un componente se describe por medio de un conjunto de roles. Tendremos un rol por cada una de las interacciones que dicho componente establezca con el resto del sistema (es decir, con los otros componentes del sistema del que forma parte). Estos roles describen el componente desde el punto de vista de los otros componentes con los que interactúa. Cada rol es una abstracción que representa tanto el comportamiento que el componente ofrece a su entorno, como el que exige a aquéllos conectados a él. Como se verá en la sección siguiente, LEDA distingue entre clases e instancias de roles, y proporciona mecanismos para la extensión y derivación de roles.

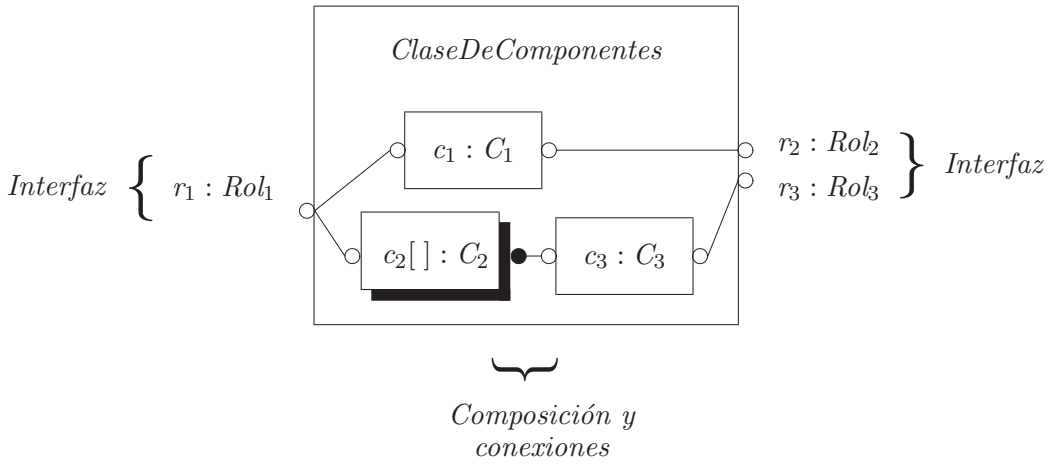


Figura 3.1: Notación gráfica para la representación de componentes.

De manera similar, la sección de composición describe la estructura interna del componente como una lista de instancias de clases de componentes, precedida de la palabra reservada **composition**. Esto permite la construcción de componentes *compuestos* a partir de otros más simples, en tantos niveles como sea necesario.

Por último, la sección de conexiones, introducida por la palabra reservada **attachments**, contiene una lista de conexiones entre roles de componentes de la sección de composición. Estas conexiones sirven para indicar cómo se construye un compuesto a partir de sus componentes constituyentes.

Al crear una instancia de un componente, se crean sus variables locales, inicializándolas bien a los valores indicados en la especificación, o bien a los valores por defecto de los tipos correspondientes, si no se ha especificado un valor inicial para ellas. Así mismo, se crean las instancias de roles que forman su interfaz, y también los subcomponentes que lo constituyen. Estos últimos se interconectan tal como se indica en la sección de conexiones.

Como es lógico, no todas las clases de componentes precisan que se detallen las cinco secciones a que hemos hecho referencia. La descripción de los componentes más simples puede realizarse especificando tan sólo su comportamiento observable, esto es, su interfaz, como un conjunto de roles independientes, sin entrar a describir su estructura interna. Sin embargo, para componentes más complejos —los compuestos—, utilizaremos también las secciones de composición y conexiones, indicando cuáles son los componentes que lo constituyen y qué conexiones se establecen entre ellos. En ocasiones, será necesario utilizar también variables locales, normalmente para coordinar los roles de la interfaz unos con otros. Tanto en el caso de las variables, como de los roles y los subcomponentes, las instancias especificadas pueden ser simples o estar organizadas en colecciones, de tamaño fijo o indeterminado, tal como veremos en los diversos ejemplos que figuran en este capítulo.

La Figura 3.1 muestra por medio de un ejemplo la notación gráfica utilizada para los componentes. Como se puede apreciar, éstos aparecen representados mediante rectángulos etiquetados, en cuyo contorno se sitúan los roles que constituyen su interfaz. Estos roles se representan mediante pequeños círculos (vacíos o rellenos, dependiendo de la forma en que se conecten unos con otros, como veremos más adelante). Si el componente se trata de un compuesto, podemos representar su estructura o composición interna, para lo cual mostraremos qué componentes lo

forman, y cómo se conectan estos componentes unos con otros. Estas conexiones se representan mediante líneas que unen los roles de dichos componentes.

Obsérvese que la representación gráfica es la misma —un rectángulo— tanto para clases de componentes como para las instancias de dichas clases. La diferencia entre ambas reside únicamente en la etiqueta. Así, la Figura 3.1 representa la clase *ClaseDeComponentes*, mientras que al indicar la estructura interna de dicha clase, estamos haciendo referencia a instancias de componentes, como se puede observar en su etiqueta, donde el nombre de la clase correspondiente viene precedido de un nombre de instancia seguido por dos puntos. De esta manera, la figura indica que la clase *ClaseDeComponentes* está compuesta por una instancia —denominada c_1— de una clase C_1, una colección de instancias —el vector $c_2[\,]$— de una clase C_2 y una instancia —c_3— de la clase C_3. Lo mismo sucede con los roles, tanto componentes como instancias se representan mediante un círculo. En la figura, estamos indicando que la interfaz de la clase *ClaseDeComponentes* está formada por instancias —denominadas r_1,r_2 y r_3— de las clases de roles Rol_1, Rol_2 y Rol_3, respectivamente. Por último, las líneas entre los roles de la Figura 3.1 representan las conexiones que conforman la estructura de la *ClaseDeComponentes* y permiten construir instancias de esta clase a partir de sus subcomponentes constituyentes, así como también indican la interfaz de la clase, por medio de la conexión de algunos de los roles de los subcomponentes con el exterior.

En este capítulo abordaremos uno a uno todos estos elementos, ilustrándolos con diversos ejemplos. Comenzaremos por mostrar la especificación en LEDA de dos componentes muy simples:

Ejemplo 3.1 *Consideremos la transmisión de un flujo de datos entre dos componentes, que llamaremos respectivamente* Emisor *y* Receptor*. Cada uno de ellos juega un único papel en el sistema, por tanto su interfaz estará constituida por un solo rol. Así, el componente* Receptor *juega el rol de* lector*, recibiendo los datos que le son enviados por el* Emisor*, que a su vez actúa como* escritor*. La especificación textual y gráfica en LEDA de estos componentes se muestra en la Figura 3.2:*

```
component Emisor {
   interface
      escritor : Escritor;
   }
```

```
component Receptor {
   interface
      lector : Lector;
   }
```

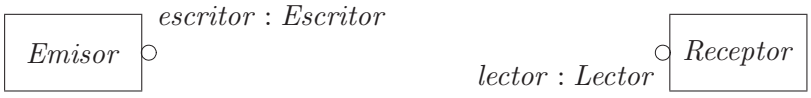


Figura 3.2: Componentes *Emisor* y *Receptor*.

donde Escritor *y* Lector *representan clases de roles. La especificación de estos roles se puede encontrar en el Ejemplo 3.2.*

3.3 Descripción del comportamiento de los componentes

Tradicionalmente, la descripción de interfaces ha estado limitada a la signatura de los métodos que exporta un componente, o a los mensajes que es capaz de recibir. En ocasiones, se indican además los métodos que el componente importa o los mensajes que éste emite, pero en cualquier caso, se especifican únicamente signaturas. Sin embargo, nuestro objetivo es describir el comportamiento observable de los componentes, es decir, cómo reaccionan a los estímulos externos, y cómo se relacionan los estímulos de entrada y de salida. Este comportamiento se describe en los roles que forman la interfaz de los componentes. Como ya hemos visto en el Capítulo 2, los roles se especifican mediante procesos en el cálculo π.

3.3.1 Notación

La notación que utilizaremos en este capítulo deriva de la original del cálculo π, en su versión poliádica, descrita en la Sección 2.1. Hemos modificado ligeramente dicha notación para poder incluirla dentro de la sintaxis más amplia del lenguaje LEDA. Las diferencias se limitan a lo siguiente:

t representa una acción interna τ.

$x!(y)$ representa una acción de salida, a través del enlace x, de un nombre o una secuencia de nombres y, lo que en la notación habitual del cálculo π se indica como $\overline{x}y$.

$(y)x!(y)$ representa una acción de salida ligada, consistente en la emisión, a través del enlace x, de un nombre o secuencia de nombres privados y, lo que en la notación habitual del cálculo π se indica como $(y)\overline{x}y$, en forma expandida, o bien $\overline{x}(y)$ en forma abreviada. En LEDA, las acciones de salida ligadas se indican siempre en su forma expandida.

$x?(z)$ representa una acción de entrada, a través del enlace x, de un nombre o una secuencia de nombres que sustituirán a z en el proceso receptor, lo que en la notación habitual del cálculo π se indica como $x(z)$.

$x[*]!(y)$ representa la difusión, a través del enlace x, de un nombre o secuencia de nombres y, al resto de los procesos que comparten ese nombre de canal. La realización de este tipo de difusiones está asociada al uso de enlaces 1:M en las conexiones múltiples, tal como se verá en la Sección 3.5.3. La traducción de estos enlaces 1:M al cálculo π estándar pasa por el uso de M+1 enlaces auxiliares, tal como se muestra en la Sección 3.9.1.

$x[\,]?(z)$ representa la acción complementaria a la difusión, siendo por lo demás equivalente a la acción de entrada $x?(z)$.

$x!()$, $x?()$ representan respectivamente un evento de salida y otro de entrada a través del enlace x, es decir equivalen a $\overline{x}$ y x.

Aparte de estas modificaciones meramente sintácticas, hemos introducido algunos otros cambios que hacen menos engorroso el uso del cálculo. Así, y aunque el cálculo π estándar no proporciona tipos de datos predefinidos, éstos pueden ser modelados fácilmente. Por este motivo, nos permitimos la libertad de utilizar constantes y variables de tipos de datos básicos, tales como *Integer*, *Boolean*, etc., así como operaciones de asignación, aritméticas y de comparación habituales. Existen diversas formas de realizar la codificación en el cálculo π de estos tipos de datos básicos, tal como se muestra en [Milner et al., 1992]. El objetivo de estas extensiones no

está en especificar las computaciones que los componentes realizan con los datos que reciben (lo que formaría parte de su implementación), sino sólo en proporcionar una información más detallada sobre su protocolo de comportamiento, es decir, sobre su interfaz, en aquellos casos en que sea necesario.

Por último, la notación que usamos permite especificar la creación, en un punto determinado del protocolo de un rol, de un componente, lo que se indica mediante la palabra reservada **new** seguida del nombre de la instancia de componente creada:

(**new** *componente*)

desde el punto de vista del cálculo π esta acción puede considerarse simplemente como la creación de un nuevo nombre *componente*, privado al agente donde se creó. La referencia al nuevo componente puede ser enviada a través de un enlace, del mismo modo que se puede ampliar el ámbito de un nombre privado. En el Ejemplo 3.3 se muestra esta forma de creación dinámica y transmisión de componentes.

3.3.2 Especificación de roles

En el Capítulo 2 figuran diversos ejemplos de roles que describen el comportamiento de sus respectivos componentes. En esta sección, únicamente pretendemos mostrar la sintaxis de LEDA para la especificación de roles. Esta especificación puede indicarse inmersa dentro del propio componente, o realizarse de forma separada, como una clase de roles con entidad propia.

La declaración inmersa de una clase de roles se realiza dentro de la sección de interfaz del componente, justo a continuación del nombre de la instancia de rol, indicando cuáles son sus *nombres libres* o parámetros que servirán de canales de comunicación para su conexión con otros roles, y seguido de la especificación de su comportamiento por medio del cálculo π.

component *ClaseDeComponentes* {
 interface
  instanciaDeRol(*parámetros*) {
   especificación del rol;
   }
  ...

Una declaración de este tipo define una clase de roles anónima a partir de la identificación de una o varias instancias de dicha clase. Esta declaración anónima no permite la reutilización ni la extensión de la clase descrita en otros componentes y arquitecturas, ni tampoco el uso de recursión en la especificación del rol, por lo que es más común el uso de un identificador de clase:

component *ClaseDeComponentes* {
interface
instanciaDeRol : ClaseDeRoles(*parámetros*) {
especificación del rol;
}
...

Existe, por último, una forma adicional de realizar la declaración de una clase de roles de forma inmersa en un compuesto, que consiste en definirla de forma implícita a partir de los roles de algunos de sus componentes. Mostraremos el uso de esta definición implícita en la Sección 3.5.4.

En los dos últimos casos, para hacer referencia posterior a una clase de roles definida de forma inmersa en un componente, indicaremos el nombre de la clase de componentes seguido del nombre de la clase de roles, separados por un punto:

ClaseDeComponentes.ClaseDeRoles

Por el contrario, cuando se realiza la especificación independiente de una clase de roles, ésta va precedida de la palabra reservada **role**, seguida a continuación del nombre de la clase, la lista de nombres libres del rol, y la especificación de su comportamiento en el cálculo π, tal como se muestra a continuación:

role *ClaseDeRoles*(*parámetros*) {
constant
listaDeConstantes;
var
listaDeVariables;
spec is
especificación de agente;
agent *NombreDeAgente* **is**
especificación de agente;
...
}

No obstante la forma en que se haya realizado la declaración de una clase de roles —independiente o inmersa, anónima o no—, la especificación de su comportamiento consistirá en la definición de uno o varios agentes o procesos utilizando la notación para el cálculo π descrita anteriormente en la Sección 3.3.1, delimitada entre llaves. La terminación de la especificación de cada uno de los agentes se indica mediante punto y coma.

En caso de que el comportamiento del rol venga descrito por un único agente, la especificación de éste figurará a continuación de las palabras reservadas **spec is**. No se indicará el nombre del agente ni sus parámetros, puesto que coincidirán con el nombre y parámetros de la clase de rol. Sin embargo, para roles más complejos, que describan un comportamiento que

pasa por varios estados distintos y que, por tanto, precise de varios agentes para su especificación, será el agente que represente el estado inicial del rol el que figure en primer lugar, tras las palabras **spec is**, mientras que la descripción del resto irá precedida por la palabra reservada **agent**. En este caso, será necesario indicar además el nombre y parámetros de estos otros agentes, seguidos de la palabra reservada **is**. A continuación figurará la especificación del comportamiento del agente, indicada mediante un proceso en el cálculo π.

Al igual que sucede con los componentes, dentro de la especificación de los roles pueden figurar declaraciones de constantes y variables locales, precedidas respectivamente por las palabras reservadas **constant** y **var**, a las que se hará referencia en la especificación de los agentes. En caso de existir varios agentes en el rol, estas variables pueden ser utilizadas para compartir información entre ellos sobre el estado del rol. Obsérvese cómo el uso de estas variables es sólo una abreviatura sintáctica de la notación original del cálculo; se puede prescindir fácilmente de esta declaración de variables y constantes con sólo añadirlas a la lista de nombres libres de cada uno de los agentes del rol.

Los roles *Lector* y *Escritor* que figuran a continuación especifican el protocolo de interacción entre los componentes *Emisor* y *Receptor* a los que hacía referencia el Ejemplo 3.1. Estos roles describen, por tanto, cómo se comportan dichos componentes con objeto de realizar con éxito una transmisión de datos.

Ejemplo 3.2 *Los datos se transmiten entre el* Lector *y el* Escritor *al emparejar la acción de salida* —write!(dato)— *del* Escritor *con la de entrada* —read?(dato)— *del* Lector *(el renombramiento de los enlaces* write *y* read*, de forma que ambas acciones puedan sincronizar, se realizará cuando se conecten ambos roles, como veremos más adelante en el Ejemplo 3.8). Tal como indica el uso de decisiones locales (indicadas por la combinación de transiciones τ y el operador suma), la responsabilidad de actuar recae en el* Escritor*, que es quien conoce cuándo el flujo de datos ha sido transmitido completamente, y quien envía un evento* eot!() *(*end of transmission*, fin de la transmisión) para notificarlo, mientras que el* Lector *ha de ser capaz de reaccionar ante cualquiera de las dos acciones del* Escritor*, por lo que sus dos alternativas se especifican en forma de decisión global.*

```
role Escritor(write,eot) {
  spec is
    t.(dato)write!(dato).Escritor(write,eot)
   + t.eot!().0;
  }
```

```
role Lector(read,eot) {
  spec is
    read?(dato).Lector(read,eot)
   + eot?().0;
  }
```

3.4 Compuestos y arquitecturas

Los componentes descritos en LEDA pueden ser simples, como el *Emisor* y el *Receptor* que acabamos de mostrar, o compuestos. Un compuesto contiene varios subcomponentes que son instancias de otras clases de componentes. Cualquier sistema de software puede ser descrito como un compuesto. Por tanto, la sintaxis de LEDA no distingue entre componentes y sistemas completos, todos se describen como clases de componentes.

Tal como hemos visto en el Ejemplo 3.1, los componentes simples vienen especificados por los roles de sus interfaces, pero para los compuestos, debemos indicar también su arquitectura

interna. Esta arquitectura es el resultado de la interconexión de varios subcomponentes. Por tanto, en la especificación de un compuesto, figurará no sólo la sección **interface**, sino también las secciones **composition** y **attachments** y, cuando sea necesario, declaraciones de constantes y variables.

Mostraremos la especificación de compuestos en LEDA por medio de una serie de ejemplos de complejidad creciente, que describen una familia de sistemas que siguen el estilo arquitectónico Cliente/Servidor.

Ejemplo 3.3 *Consideremos en primer lugar un sistema Cliente/Servidor muy simple, en el cual un cliente realiza sucesivas peticiones de servicios a un servidor. Cada vez que recibe una petición, el servidor crea un componente para realizar el servicio —componente al que denominaremos* sirviente*, siguiendo la terminología de CORBA— y envía su referencia al cliente. La especificación en LEDA de estos componentes, junto con los roles que definen su interacción (uno definido de forma inmersa y otro de forma separada), es la siguiente:*

```
component Cliente {
  interface
    pedir: Pedir;
  composition
    sirviente[ ] : any;
  }

role Pedir(petición) {
  spec is
    (respuesta)petición!(respuesta).
      respuesta?(sirviente).Pedir(petición);
  }
```

```
component Servidor {
  interface
    servir: Servir(petición) {
      var
        n : Integer := 0;
      spec is
        petición?(respuesta).
          (new sirviente)respuesta!(sirviente).
          n++.Servir(petición);
      }
  composition
    sirviente[ ] : any;
  }
```

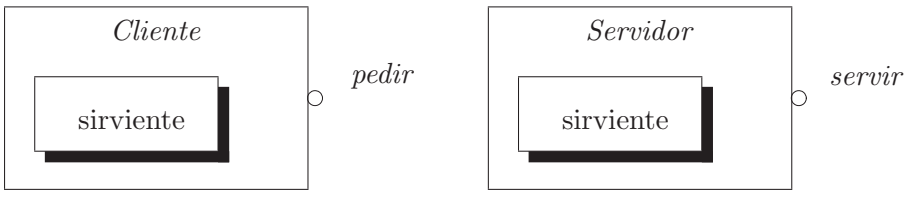


Figura 3.3: Compuestos *Cliente* y *Servidor*.

Como se puede observar, tanto el Cliente *como el* Servidor *son compuestos que están constituidos por un número indeterminado de componentes* sirviente*, representados por sendas colecciones no limitadas.*

La interfaz del Cliente *está formada por un único rol* pedir*. La clase de roles* Pedir *describe el comportamiento de este rol (y por tanto el del cliente), mientras que el comportamiento del*

Servidor *viene determinado por el rol* servir*, que hemos definido inmerso en la especificación de este componente.*

Tal como indica la especificación de la clase de roles Pedir*, el cliente realiza sus peticiones realizando una acción de salida a través del enlace* petición*. Como objeto o argumento de dicha acción de salida figura el nombre del enlace de respuesta por el que se recibirá posteriormente el* sirviente *(o más precisamente, una referencia al mismo). Este nombre* respuesta *es creado a tal efecto, de manera que se trata de una acción de salida ligada.*

Al recibir una petición, el Servidor *crea un componente* sirviente *mediante la instrucción* **new***. A continuación, la referencia al* sirviente *se transmite al componente* Cliente *a través del canal* respuesta *que se acaba de recibir y que constituye un enlace privado entre el servidor y el cliente. Obsérvese que de esta forma hacemos uso de la movilidad proporcionada por el cálculo* π *para establecer canales de respuesta privados para cada petición, lo que será aprovechado en ejemplos posteriores para conectar varios clientes a un mismo servidor a través de un único rol* pedir*.*

La variable n *se utiliza en el rol* servir *para llevar la cuenta del número de peticiones recibidas, lo que también será utilizado en un ejemplo posterior.*

La especificación no indica el tipo del componente sirviente*, que es declarado en ambos componentes como perteneciente a un tipo genérico* **any***, lo que permitirá el posterior refinamiento de esta arquitectura Cliente/Servidor, tal como se mostrará en la Sección 3.7, para proporcionar diferentes servicios.*

Como acabamos de ver, una clase de componentes en LEDA especifica la arquitectura de un grupo de componentes de software, ya sean estos componentes simples, subsistemas, o incluso sistemas completos. En el caso de los componentes simples y subsistemas, nos referiremos a instancias de dichas clases en la sección **composition** de otros componentes, para formar así compuestos cada vez más complejos. En el caso de una clase que represente la arquitectura de un sistema completo, podremos obtener un ejemplar de dicha arquitectura —es decir, un sistema— mediante la *instanciación* de la misma. Así:

instance *miSistema : Sistema;*

representa una instancia, denominada *miSistema* de la arquitectura descrita por la clase de componentes *Sistema*. La instanciación de arquitecturas está relacionada con la obtención de prototipos ejecutables a partir de su especificación en LEDA, tema que trataremos en el Capítulo 4.

3.5 Conexiones

La arquitectura de un compuesto viene determinada por las relaciones que sus componentes mantienen entre sí. Estas relaciones se representan de forma explícita en LEDA por medio de una serie de conexiones entre los roles de los componentes del compuesto. Las conexiones relacionan roles de componentes distintos, y se especifican en la sección **attachments** del compuesto que los contiene. Las conexiones se establecen cuando se crean las correspondientes instancias de componentes y roles, posiblemente de forma dinámica, y pueden ser modificadas a lo largo de la ejecución del sistema.

LEDA distingue entre varios tipos de conexiones, lo que permite la especificación tanto de arquitecturas estáticas, como reconfigurables y dinámicas. Como mostraremos más adelante,

esto representa una novedad frente a otras propuestas anteriores que permiten únicamente la descripción de arquitecturas inmutables.

3.5.1 Conexiones estáticas

Las *conexiones estáticas* son aquéllas que no se modifican nunca una vez que se establecen, y sirven para describir, por tanto, sistemas que tienen una arquitectura estática. La sintaxis de una conexión estática entre dos roles es la siguiente:

$$comp.rol(nombres) \ <> \ comp'.rol'(nombres');$$

que indica que el rol *rol* del componente *comp* se conecta al rol rol' del componente $comp'$. Los nombres de parámetros *nombres* y $nombres'$ deben coincidir en su conjunto, aunque se permiten variaciones en el orden en que están indicados en la conexión, como veremos posteriormente. La representación gráfica de una conexión estática consiste en una línea continua que une los dos roles conectados, tal como muestra la Figura 3.4.

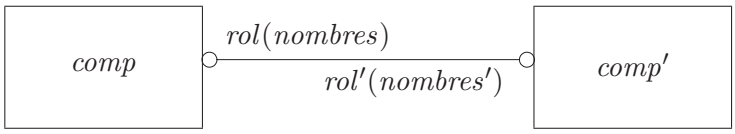


Figura 3.4: Conexión estática entre dos roles.

Tanto *comp* como $comp'$ deben estar declarados en la sección **composition** del compuesto donde figura la conexión.

Ejemplo 3.4 *Consideremos de nuevo los componentes* Cliente *y* Servidor *del Ejemplo 3.3. Podemos especificar la arquitectura de un sistema Cliente/Servidor por medio de un compuesto que contenga ambos componentes y que conecte sus roles utilizando una conexión estática, tal como figura a continuación.*

```
component ClienteServidor {
  interface none;
  composition
    cliente : Cliente;
    servidor : Servidor;
  attachments
    cliente.pedir(p) <> servidor.servir(p);
  }
```

Para referirnos a una instancia concreta de la clase ClienteServidor procederemos a su instanciación:

```
instance cs : ClienteServidor;
```

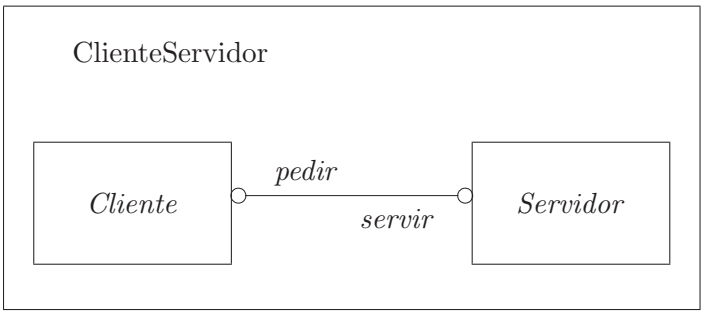


Figura 3.5: Sistema Cliente/Servidor.

Con respecto a los nombres libres de cada uno de los roles conectados, el uso de los mismos nombres de enlace en los términos izquierdo o derecho de la conexión determina la sustitución de nombres que se realiza al conectar los roles. Así, por ejemplo, si tenemos las instancias de rol:

$$rol_a : Rol_a(a_1, a_2, a_3) \quad y \quad rol_b : Rol_b(b_1, b_2, b_3)$$

cuyos nombres libres son diferentes, la conexión

$$rol_a(n_1, n_2, n_3) <> rol_b(n_2, n_3, n_1)$$

determina el renombramiento de los nombres de enlace que figuran en las clases Rol_a y Rol_b, mediante las sustituciones $\{n_1/a_1, n_2/a_2, n_3/a_3\}$ y $\{n_2/b_1, n_3/b_2, n_1/b_3\}$ de forma que se produce la identificación de los nombres a_1 con b_3, a_2 con b_1 y a_3 con b_2.

No obstante esta posibilidad de renombramiento, y con objeto de permanecer en el marco formal establecido por la Definición 2.26 de agentes compatibles (que establece una distinción entre los nombres libres de los roles), los nombres de enlace usados como parámetros reales en una conexión han de ser diferentes dentro de cada uno de los roles, de forma que no consideraremos aceptables conexiones del tipo:

$$rol_a(n_1, n_2, n_3) <> rol_b(n_2, n_1, n_1)$$

puesto que realizan de forma implícita la identificación de los nombres de enlace b_2 y b_3 del rol de clase Rol_b.

Por último, es posible conectar más de dos roles en la misma conexión, tal como muestra la Figura 3.6. Al igual que en los casos anteriores, la regla para realizar estas conexiones consiste únicamente en que el conjunto de nombres libres de los roles del término izquierdo debe de coincidir con el de los que figuran en el término derecho, sin identificar nombres de enlace dentro del mismo rol. Así, por ejemplo:

$$comp.rol(p_1, p_2) <> comp'.rol'(p_1, p_3) <> comp''.rol''(p_2, p_3);$$

representa la conexión de roles de tres componentes distintos, cuyos nombres libres se instancian de acuerdo a las identificaciones que se realizan en la conexión, que en este caso indica que el rol *comp.rol* se conecta al rol $comp'.rol'$ a través del primer enlace de cada uno de ellos (el denominado p_1 en la conexión), mientras que ambos roles se conectan a $comp''.rol''$ a través del segundo enlace de cada uno de ellos (los denominados respectivamente p_2 y p_3).

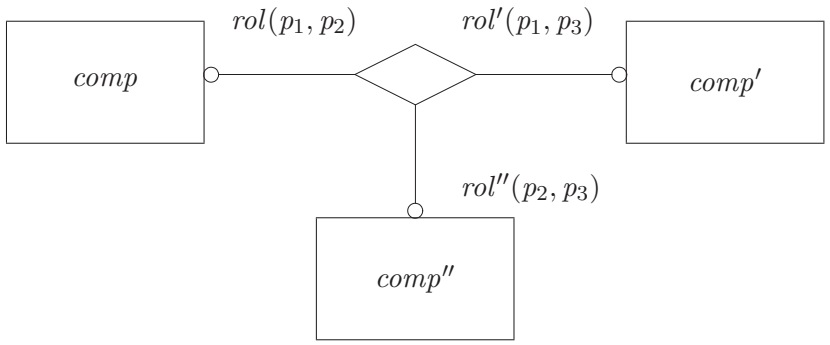


Figura 3.6: Conexión estática entre más de dos roles.

3.5.2 Conexiones reconfigurables

Por su parte, las *conexiones reconfigurables* se utilizan en arquitecturas que presentan varias configuraciones, es decir, en las que los patrones de interconexión entre componentes varían en el tiempo, y en que los roles conectados dependen de ciertas condiciones.

Una conexión reconfigurable es aquélla en la que al menos uno de sus términos es una expresión condicional, lo que se representa con la siguiente sintaxis:

$$comp.rol(nombres) \text{ <> } \textbf{if } condición \\ \textbf{then } comp'.rol'(nombres') \\ \textbf{else } comp''.rol''(nombres'');$$

que indica que el rol *rol* del componente *comp* se conecta bien al rol rol' del componente $comp'$ cuando *condición* toma el valor lógico TRUE, o bien al rol rol'' del componente $comp''$ cuando *condición* toma el valor lógico FALSE. Las conjuntos de nombres libres *nombres*, $nombres'$ y $nombres''$ deben coincidir, aunque puede haber variaciones en su ordenación.

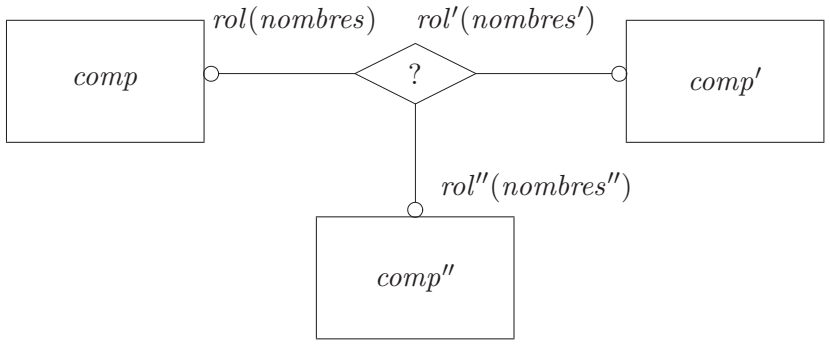


Figura 3.7: Conexión reconfigurable.

La condición que figura en el segundo término de la conexión, tras la palabra reservada **if**, debe ser una expresión lógica en función de variables y constantes declaradas en el compuesto donde figura la conexión o en alguno de sus subcomponentes. Esta condición se evalúa cada vez que el rol conectado en el término no condicional realiza un cambio de estado volviendo a su estado inicial (es decir, al agente identificado implícitamente con el nombre del rol).

En caso de que en la conexión se puedan presentar más de dos alternativas, ésta se indica mediante un término condicional **case**. Así, por ejemplo:

$$
\begin{aligned}
comp.rol(nombres) <> \ & \textbf{case } condición' \quad \textbf{then } comp'.rol'(nombres') \\
& \textbf{case } condición'' \quad \textbf{then } comp''.rol''(nombres'') \\
& \ldots \\
& \textbf{default } comp_n.rol_n(nombres_n);
\end{aligned}
$$

En el supuesto de que las condiciones de cada una de las alternativas no sean excluyentes, se seleccionará una de las que tomen valor lógico TRUE, de forma no determinista.

Ejemplo 3.5 *Supongamos ahora que tenemos dos componentes* Servidor*, y que cada petición se asigna a uno u otro, tratando de equilibrar su carga de trabajo. La especificación de este compuesto es la indicada en la Figura 3.8*

```
component ClienteServidorReconfigurable {
  interface none;
  composition
    cliente : Cliente;
    servidor[2] : Servidor;
  attachments
    cliente.pedir(p) <> if ( servidor[1].n <= servidor[2].n )
                          then servidor[1].servir(p)
                          else servidor[2].servir(p);
  }
```

Para obtener a un ejemplar de la arquitectura ClienteServidorReconfigurable*, procederemos a su instanciación:*

```
instance csr : ClienteServidorReconfigurable;
```

3.5.3 Conexiones múltiples

Por último, las *conexiones múltiples* describen patrones de comunicación entre vectores o colecciones de componentes. Cada par de componentes interconectados puede usar enlaces privados en su comunicación, o bien estos enlaces pueden estar compartidos entre todos los componentes implicados. Por tanto, las conexiones múltiples pueden ser *compartidas* o *privadas*.

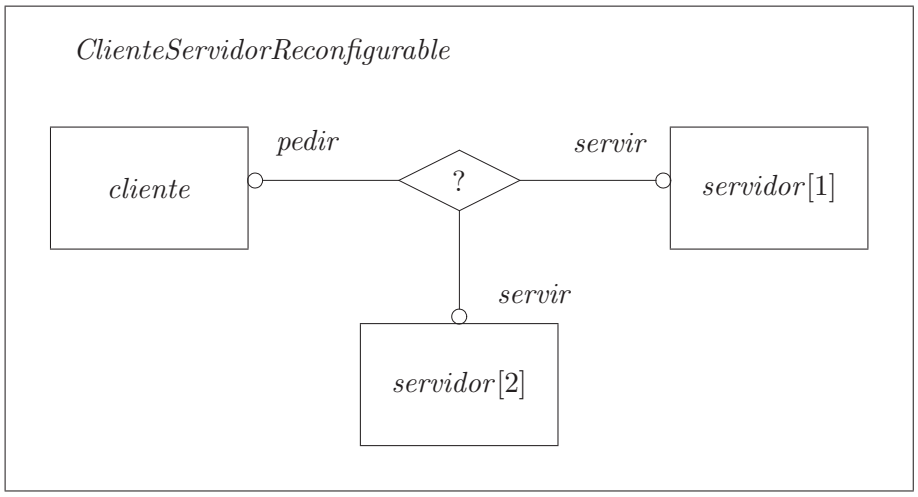


Figura 3.8: Sistema Cliente/Servidor reconfigurable.

La sintaxis de una conexión múltiple que utilice nombres privados es la misma que la de una conexión estática, exceptuando el que, en cada término, en vez de una única instancia de componente y rol, figura al menos una colección. Así, por ejemplo:

$$comp[\,].rol(nombres) <> comp'.rol'[\,](nombres');$$

indica que el rol *rol* de cada uno de los componentes de la colección *comp* se conecta a uno de los roles de la colección rol' del componente $comp'$, tal como se ve en la Figura 3.9. Para cada par de roles conectado se utiliza una copia privada de los nombres libres o parámetros de los roles. Las conexiones se establecen de forma dinámica, según se crean los correspondientes componentes y roles que figuran en cada término.

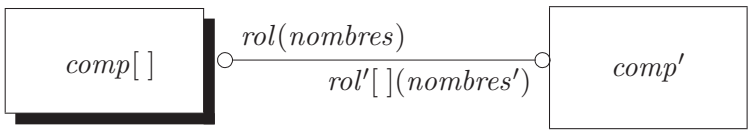


Figura 3.9: Conexión múltiple que utiliza nombres privados.

Por el contrario, el uso de un asterisco en alguna de las colecciones que figuren en una conexión múltiple, indica que todos sus roles van a utilizar nombres compartidos a la hora de conectarse con el rol o roles que figuren en el otro término. Así, por ejemplo:

$$comp[*].rol(nombres) <> comp'.rol'(nombres');$$

indica que los roles *rol* de todos los componentes de la colección *comp* se conectan al mismo rol rol' del componente $comp'$, a medida que se van creando los primeros, y usando siempre los mismos nombres compartidos para los parámetros.

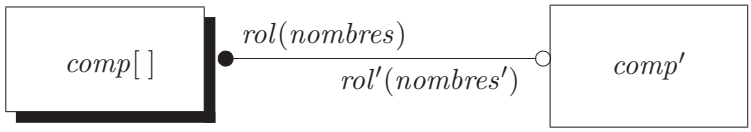


Figura 3.10: Conexión múltiple con nombres compartidos.

Por tanto, una conexión múltiple compartida implica el uso de una serie de canales de comunicación 1:M (uno por cada nombre libre de los roles) y la realización de una difusión cada vez que se realice una acción de salida desde el extremo sencillo del canal hacia el extremo múltiple. Por el contrario, una conexión privada establece múltiples canales de comunicación 1:1, uno por cada par de roles conectados y por cada nombre libre de dichos roles.

Ejemplo 3.6 *Consideremos ahora un sistema Cliente/Servidor más realista, en el cual varios clientes están conectados a una batería de servidores, tal como se especifica a continuación.*

component *BateríaDeServidores* {
interface
servir(*p*) {
spec is implicit;
}
composition
servidor[ ] : *Servidor*;
attachments
servidor[∗].*servir*(*p*) >> *servir*(*p*);
}

component *ClienteServidorMúltiple* {
interface none;
composition
cliente[ ] : *Cliente*;
batería : *BateríaDeServidores*;
attachments
cliente[∗].*pedir*(*p*) <> *batería.servir*(*p*);
}

El componente BateríaDeServidores*, compuesto por un número indeterminado de componentes* Servidor*, presenta una interfaz indicada mediante un rol* servir*. Tal como se observa en la especificación textual anterior, este rol no se describe de forma explícita mediante uno o*

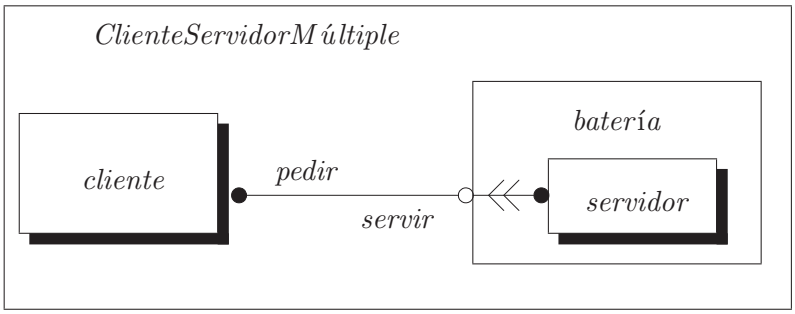


Figura 3.11: Sistema Cliente/Servidor con varios clientes y batería de servidores.

varios agentes (como es el caso de todos los roles presentados hasta este momento), sino que dicha definición se realiza de forma implícita en la sección de conexiones mediante la reunión (en una conexión múltiple compartida) de los roles servir *de cada uno de los componentes* Servidor*. La compartición del nombre de enlace* p *implica que cada petición será atendida por uno cualquiera de los servidores, de manera no determinista. Según la citada conexión, el agente que especifica al rol* servir *en el cálculo* π *consiste en la composición paralela de los roles* servir *de cada uno de los componentes, que de esta manera, son exportados por la batería de servidores (la exportación de roles (indicada por el operador* >>*) se describe a continuación, en la Sección 3.5.4).*

Por otra parte, la conexión entre los clientes y la batería de servidores en el compuesto ClienteServidorMúltiple *es también múltiple, y todos los clientes comparten el enlace* p *a través del cual solicitan sirvientes. Sin embargo, tal como vimos en el Ejemplo 3.3, donde figura la especificación de los roles* pedir *y* servir *a que se hace referencia aquí, el canal de respuesta utilizado es exclusivo para cada petición, lo que asegura que el sirviente llegue al cliente que lo solicitó, aunque todos los clientes estén conectados a la batería de servidores a través de un único canal de petición* p*.*

De nuevo, para obtener un ejemplar de la clase ClienteServidorMúltiple procederemos a instanciar la arquitectura:

instance *csm : ClienteServidorMultiple;*

Un ejemplo como el que acabamos de describir, difícilmente puede ser especificado por medio de formalismos como CSP o CCS, lo que muestra la mayor expresividad del cálculo π (y por ende también de nuestro lenguaje) frente a otras álgebras de procesos y a otros ADLs derivados de ellas. En efecto, por medio de ADLs basados en CSP (como es el caso de Wright) únicamente es posible simular el uso de conexiones reconfigurables, como la descrita en el Ejemplo 3.5 de la sección anterior, pero no múltiples, como la que nos ocupa, puesto que al no existir posibilidad de crear nuevos nombres, ni de transmitir nombres entre procesos, todas las configuraciones posibles del sistema han de estar previstas de antemano, y han de basarse en el uso de nombres también predeterminados, tal como los propios autores de Wright muestran en [Allen et al., 1998].

Por el contrario, la movilidad que proporciona el cálculo π permite la creación de nuevos nombres y su transmisión entre procesos, lo que ofrece posibilidades adicionales de configuración, como es el caso del ejemplo que acabamos de presentar, donde se usa un canal múltiple para el envío de mensajes (con lo cual el receptor no conoce cuál es el emisor que lo ha realizado), junto con canales específicos para la respuesta (lo que permite que ésta sea pueda ser transmitida en exclusiva a quien realizó la emisión). Aparte de su mayor expresividad, las especificaciones en LEDA resultan también más sencillas e inteligibles, al utilizar un menor número de nombres de enlace. En ejemplos posteriores, seguiremos haciendo uso de estas características particulares de nuestro lenguaje que lo convierten en una opción ventajosa frente a otras propuestas en este campo.

3.5.4 Exportación de roles

Una forma adicional de conexión es la exportación de roles. Por lo general, al especificar un compuesto no todos los roles de sus componentes se utilizan para interconectar dichos componentes unos con otros, sino que alguno de ellos pasa a formar parte de la interfaz del

compuesto. En ese caso, diremos que estos roles son exportados por el compuesto, lo que se indica en LEDA por medio del operador >> en lugar de <>.

En el primer término de una conexión de exportación pueden figurar varias instancias de roles, separadas por comas; en el segundo término sólo puede aparecer un rol:

$$comp'.rol'(nombres'),$$
$$comp''.rol''[\,](nombres'') \texttt{ >> } rol(nombres);$$

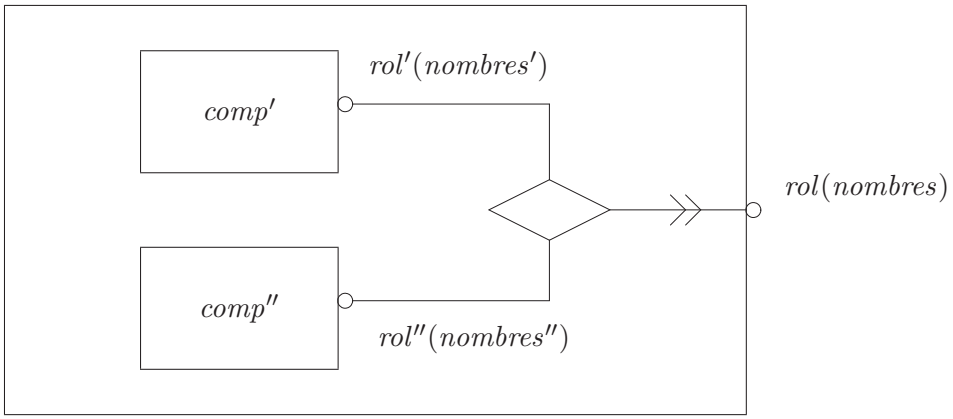


Figura 3.12: Exportación de roles.

Los roles del primer término formarán parte de la interfaz de los componentes del compuesto donde figure la conexión de exportación, mientras que el rol del segundo término habrá de formar parte de la interfaz de dicho compuesto. Respecto al uso de nombres de enlace en la conexión de exportación, ésta sigue las mismas reglas que el resto de conexiones.

La exportación de roles implica la definición, de forma inmersa e implícita, de una clase de roles. Por lo general, esta clase de roles es anónima. Consideraremos que la especificación de una clase de roles así definida consiste en la composición en paralelo de los roles exportados, con las identificaciones de nombres de enlaces que indique la conexión donde se define la exportación.

Ya hemos utilizado este mecanismo de exportación en el Ejemplo 3.6, donde los roles *servir* de los componentes *Servidor* se reunían y exportaban para formar el rol *servir* de la interfaz del compuesto *BateríaDeServidores* mediante la conexión (en este caso, exportación):

$$servidor[*].servir(p) >> servir(p);$$

en la que todos los roles *servir* de los componentes *servidor* comparten el mismo nombre de enlace p a través del cual reciben las peticiones.

Como se puede apreciar, este mecanismo de exportación de roles permite especificar la *agregación* de componentes [Box, 1998], un mecanismo muy habitual en algunos modelos de componentes (como por ejemplo COM).

3.6 Extensión de roles y componentes

Con objeto de facilitar la reutilización efectiva tanto de los componentes como de la arquitectura, se precisa de un mecanismo de redefinición y extensión de roles y componentes. En nuestro lenguaje, dicho mecanismo está basado en las relaciones de herencia y de extensión de roles descritas en la Sección 2.4 y que aseguran la preservación de la compatibilidad por parte de los roles derivados.

La extensión de roles puede utilizarse para redefinir —parcial o totalmente— el rol antecesor, proporcionando una nueva especificación para alguno de sus agentes; o para extender un rol, dotándolo de funcionalidad adicional. En ambos casos es necesario comprobar, utilizando la relación de herencia que el rol extendido es, efectivamente, descendiente del rol progenitor. Esto se hará durante el análisis de la especificación, y es lo que permite el reemplazamiento de componentes y el consiguiente refinamiento de la arquitectura al que nos referiremos en la Sección 3.7.

La sintaxis completa de la extensión de roles en LEDA figura en el Apéndice A, pero puede ser presentada informalmente de la siguiente manera:

role *ClaseDerivada*(*parámetros*)
  extends *ClaseProgenitora* {
| **spec is** *NuevaEspecificación;*
| **redefining** *AgenteProgenitor* **as** *NuevaEspecificación;*
| **redefining** *AgenteProgenitor* **adding** *NuevaEspecificación;*
...
}

Así, tras el nombre de la nueva clase de rol, se indica cuál es la clase de la que se deriva, precedida de la palabra reservada **extends**. Dependiendo del carácter de esta derivación, la especificación del nuevo rol consistirá en la descripción completa del mismo (utilizando para ello una cláusula **spec is**), seguida de la especificación del agente o agentes que describen el rol), o bien en la redefinición de algunos de los agentes del progenitor. A su vez, la redefinición de un agente puede ser completa (lo que será indicado mediante la cláusula **redefining as**) o puede consistir únicamente en la adición de un comportamiento alternativo al ya descrito por el agente (indicado en este último caso mediante la cláusula **redefining adding**). La limitación de la extensión de roles a estos tres mecanismos está motivada por la simplificación de la comprobación formal de la relación de herencia, en concreto para que se cumplan las condiciones del Teorema 2.57.

Ejemplo 3.7 *Consideremos el rol* Servir *del Ejemplo 3.3. Su comportamiento puede ser extendido permitiendo a los clientes consultar el número de peticiones resueltas por el servidor, lo que podría ser utilizado, por ejemplo, con fines estadísticos, o para que los clientes decidiesen a qué servidor realizar su petición. La extensión descrita es la siguiente:*

```
role ServirConEstadísticas(petición,estadísticas)
   extends Servidor.Servir {
 redefining Servir(petición) adding
   estadísticas!(n).ServirConEstadísticas(petición,estadísticas);
 }
```

El nuevo comportamiento debe considerarse como una alternativa que se añade al rol Servir*, de forma que éste presenta ahora tanto la funcionalidad de proporcionar servicios, previa petición de sus clientes, como la de informar del número de peticiones satisfechas hasta el momento. El comportamiento especificado por el rol* ServirConEstadísticas *equivale al siguiente, en donde no hemos tomado como punto de partida el rol* Servir*:*

```
role ServirConEstadísticas'(petición,estadísticas) {
 var
   n : Integer := 0;
 spec is
   petición?(respuesta).
     (new sirviente)respuesta!(sirviente).
     n++.ServirConEstadísticas'(petición,estadísticas)
+ estadísticas!(n).ServirConEstadísticas'(petición,estadísticas);
 }
```

Este concepto de extensión también puede aplicarse a los componentes. Los componentes derivados heredan la especificación de sus progenitores, incluyendo sus roles, componentes y conexiones. Un componente derivado extiende a su progenitor al añadir nuevos roles, componentes o conexiones, o al redefinir algunos de los especificados en éste. En caso de redefinición de un rol o componente, la instancia redefinida debe ser, a su vez, heredera de la original.

La extensión de componentes se indica, de forma análoga a la de roles, mediante la cláusula **extends**, tal como muestra la Figura 3.13, y al especificar el cuerpo de la clase derivada

```
component ClaseDerivada
  extends ClaseProgenitora {
   ...
  }
```

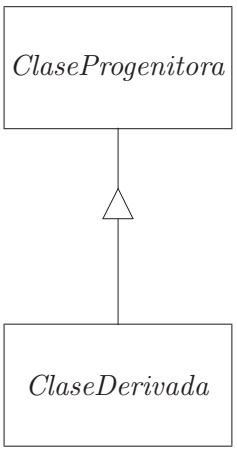


Figura 3.13: Extensión de componentes.

indicaremos cuáles son los elementos (ya sean variables, roles de la interfaz, componentes o conexiones) que añadimos o redefinimos respecto a la clase progenitora, de forma que cualquier elemento de la clase progenitora al que no hagamos referencia en la derivada se hereda en esta última tal como fue definido en la primera.

Ejemplo 3.8 *En el sistema Cliente/Servidor del Ejemplo 3.6, los sirvientes se describieron como componentes de una clase genérica* **any**. *Según esto, hemos descrito una arquitectura Cliente/Servidor abstracta que sigue un sencillo protocolo de peticiones y respuestas. Podemos describir sistemas más específicos si detallamos el servicio, especificando el comportamiento que el componente cliente y el sirviente siguen durante la prestación del servicio. Para ello, comenzaremos por describir un componente que solicita como servicio el envío de un flujo de datos, siendo por tanto* Receptor *del mismo:*

```
component ClienteReceptor extends Cliente {
  interface
    pedir : PedirEmisor(petición) extends Pedir {
      spec is
        (respuesta)petición!(respuesta).
          ( new receptor)respuesta?(sirviente).PedirEmisor(petición);
      }
  composition
    receptor[ ] : Receptor;
    sirviente[ ] : Emisor;
  attachments
    receptor[ ].lector(w,e) <> sirviente[ ].escritor(w,e);
  }
```

El componente ClienteReceptor *que acabamos de especificar es una derivación del* Cliente *del Ejemplo 3.3, tal como muestra la Figura 3.14. Su rol* pedir *se redefine indicando que se crea un componente* Receptor *cada vez que el cliente solicita un servicio. También el componente* sirviente *que presta el servicio ha sido redefinido, indicando que su tipo es ahora* Emisor*, en lugar de* **any***, y se añade una nueva conexión entre los roles del receptor y del sirviente.*

Los clases de componentes Emisor *y* Receptor *a que hacemos referencia aquí fueron especificados en el Ejemplo 3.1, mientras que sus roles* Escritor *y* Lector *figuran en el Ejemplo 3.2. Obsérvese como, a pesar de utilizar nombres de enlaces diferentes (*write *y* read*, respectivamente) en la especificación de dichos roles, la conexión que se establece en el* ClienteReceptor *indica que ambos se renombran como* w *mediante la sustitución* $\{w/write, w/read\}$.

3.7 Refinamiento de arquitecturas

Las descripciones arquitectónicas pueden utilizarse con diferentes niveles de abstracción durante el proceso de desarrollo. Esto es lo que se denomina comúnmente como refinamiento. Por ejemplo, podemos empezar la especificación de un sistema en un nivel de abstracción alto, en el que describimos únicamente sus componentes de primer nivel, las interfaces de éstos y como están interconectados para construir el sistema. A continuación, realizamos un refinamiento de

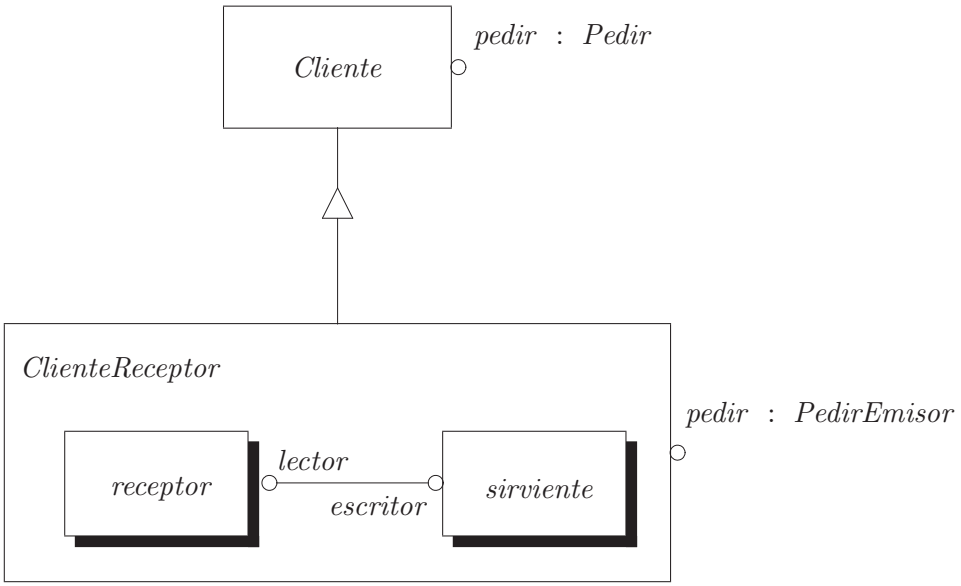


Figura 3.14: Extensión del componente *Cliente*.

la especificación con objeto de obtener una versión más detallada de la misma, describiendo la estructura interna o el comportamiento de los componentes definidos anteriormente. Se van obteniendo así especificaciones más complejas que se van acercando gradualmente a la implementación. Como hemos visto, la extensión de componentes es un mecanismo útil para el refinamiento, pero existen también otras formas de refinar especificaciones arquitectónicas.

En LEDA, la forma de llevar a cabo este refinamiento es por medio de la *instanciación* de arquitecturas, en la que es posible indicar el reemplazamiento de una instancia de componente en una arquitectura por otra instancia cuya clase extienda la de la primera. El refinamiento de arquitecturas es de gran utilidad para la especificación incremental, para la descripción de familias de productos de software con una arquitectura común, y también para la sustitución, de forma dinámica, de un componente en un sistema de software. La sintaxis utilizada para el refinamiento de una arquitectura es la siguiente:

instance *componenteDerivado : ClaseComponente* [
subcomponente : ClaseDerivadaSubcomponente;
];

que indica que *componenteDerivado* es una instancia de *ClaseComponente* en la que hemos reemplazado su *subcomponente* (que supongamos ha sido declarado como instancia de una cierta *ClaseSubcomponente*) por una instancia de *ClaseDerivadaSubcomponente*, donde *ClaseDerivada-Subcomponente* debe ser heredera de *ClaseSubcomponente* (véase la Figura 3.15).

Al refinar una arquitectura, instanciando algunos de sus componentes, se modifican además parte de las conexiones que figuran en ella, debido a que los componentes originales son reemplazados por versiones derivadas de los mismos. Sin embargo, no es necesario volver a comprobar la compatibilidad de la arquitectura, puesto que tal como se demostró en el Capítulo 2, la herencia de roles preserva la compatibilidad.

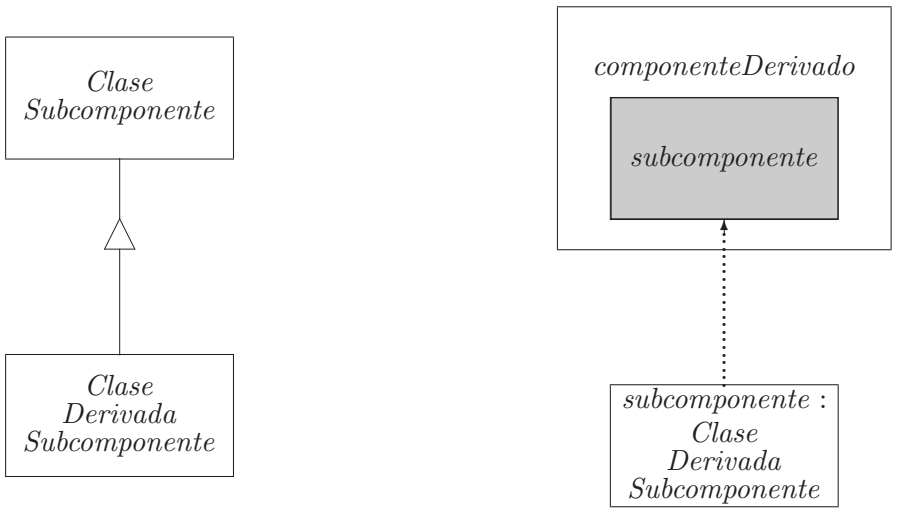


Figura 3.15: Instanciación de una arquitectura.

Ejemplo 3.9 *A partir de la clase de componentes* ClienteReceptor *descrita en el ejemplo anterior como una derivación de la clase* Cliente*, podemos refinar nuestra arquitectura Cliente/Servidor, obteniendo la descripción de un sistema en el que el servicio ofrecido es la transmisión de un fichero de datos. Un ejemplar de dicha arquitectura refinada se obtendría de la siguiente forma:*

instance *csm2 : ClienteServidorMúltiple* [
cliente[ ] *: ClienteReceptor;*
batería.servidor[ ].*sirviente*[ ] *: Emisor;*
];

que determina que csm2 *es una instancia de la arquitectura* ClienteServidorMúltiple *del Ejemplo 3.6 en la que hemos refinado los componentes* cliente *(que habían sido declarados en la arquitectura original como de clase* Cliente*) por instancias de la clase* ClienteReceptor*, derivada de ésta, así como también hemos refinado o instanciado los componentes* sirviente*, creados en la batería de servidores, indicando ahora que se trata de instancias de la clase* Emisor*, en lugar de* **any**.

3.8 Adaptadores

En ocasiones el comportamiento de dos componentes no es compatible, pero dichos componentes pueden adaptarse de forma que sean capaces de colaborar entre sí. En LEDA esto se lleva a cabo haciendo uso de un *adaptador*, que actúa como mediador entre ambos, permitiendo la construcción de compuestos a partir de componentes que no son estrictamente compatibles. Los adaptadores pueden utilizarse también para modificar la interfaz que un determinado componente exporta a su entorno.

Los adaptadores se representan gráficamente mediante un óvalo, y se especifican en el cálculo π, utilizando la misma sintaxis que los roles. Sin embargo, los roles describen la interfaz

de un componente, por lo que se declaran en la sección **interface**, mientras que los adaptadores se utilizan fundamentalmente como nexo de unión entre los componentes de un compuesto, y se declaran en la sección **composition**, junto al resto de integrantes del compuesto.

Veamos un ejemplo, continuación de los precedentes, en el que se muestra la manera de adaptar la interfaz ofrecida por un servidor no seguro a los requisitos exigidos por los clientes descritos en el Ejemplo 3.3.

Ejemplo 3.10 *En los ejemplos precedentes, los servidores están siempre listos para recibir peticiones, lo que no es una suposición demasiado realista. Un servidor no seguro, como el que se muestra en la especificación en LEDA de la Figura 3.16 (izquierda), puede fallar en cualquier momento:*

```
component ServidorNoSeguro {
 interface
  servir : ServirNS(petición,caída) {
   spec is
    t.petición?(respuesta).
     ( new sirviente)respuesta!(sirviente).
      ServirNS(petición,caída)
   + t.caída!().0;
   }
 composition
  sirviente[ ] : any;
}
```

```
component BateríaDeServidoresTF
   extends BateríaDeServidores {
 composition
  servidor[ ] : ServidorNoSeguro;
  rearrancar: Rearrancar(caída) {
   spec is
    caída?().
    (new servidor)Rearrancar(caída);
   }
 attachments
  rearrancar(e),
  servidor[∗].servir(p,e) >> servir(p);
}
```

Figura 3.16: Especificación textual de una batería de servidores tolerantes a fallos.

Obsérvese cómo por medio de decisiones locales, indicadas por la combinación del operador de suma con transiciones internas, se puede especificar que el servidor puede fallar o caer de forma inesperada, emitiendo un evento caída*.*

Como resulta evidente, el comportamiento del ServidorNoSeguro *no es compatible con el de los clientes descritos en los ejemplos anteriores, que presuponen que los servidores están siempre dispuestos a atender sus peticiones. Sin embargo, por medio de un sencillo adaptador que hemos denominado* rearrancar*, podemos construir una batería de servidores tolerante a fallos (*BateríaDeServidoresTF*, arriba a la derecha en la figura).*

Esta nueva batería está especificada como una extensión de la original BateríaDeServidores*. Dado que no se indica interfaz en la misma, debemos asumir que presenta el mismo interfaz que su progenitor (es decir, un rol* servir*, definido de forma implícita mediante la exportación de roles de sus subcomponentes). En cambio, la sección* **composition** *indica que la clase de los servidores que componen la batería se redefine como* ServidorNoSeguro *(que no es una extensión de la clase* Servidor *original), a la vez que se añade un nuevo elemento: el adaptador* rearrancar*, definido de manera inmersa. Análogamente, se modifica la sección de conexiones, indicando ahora que el rol* servir *de la batería se obtiene reuniendo no sólo los roles* servir *de los componentes, sino combinándolos también con el adaptador* rearrancar*.*

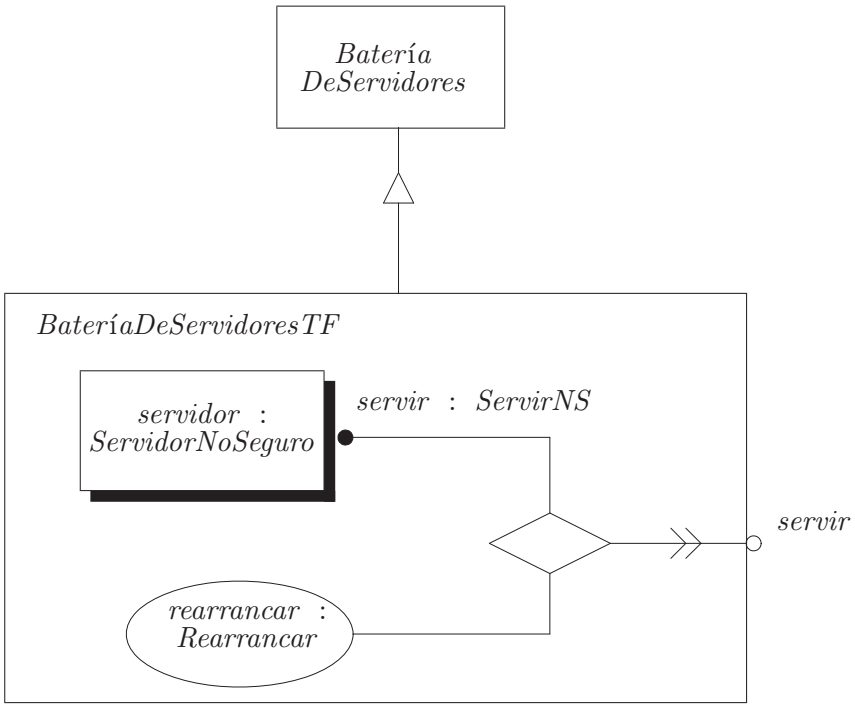


Figura 3.17: Representación gráfica de una batería de servidores tolerantes a fallos.

Por su parte, la especificación de este último elemento indica que cada vez que un servidor caiga, es rearrancado por el adaptador (que, en realidad, crea un nuevo servidor). Por tanto, el uso del adaptador modifica el comportamiento observable de la batería de servidores no seguros, y la combinación de los roles ServirNS *y el adaptador* Rearrancar *proporciona una interfaz para la* BateríaDeServidoresTF *que puede demostrarse que se deriva o hereda del rol* servir *de la batería original del Ejemplo 3.6. Según esto, podemos asegurar que la clase* BateríaDeServidoresTF *extiende la clase* BateríaDeServidores *de dicho ejemplo, y su comportamiento es compatible con el rol* Pedir *de los clientes.*

Por tanto, podemos refinar la arquitectura Cliente/Servidor del Ejemplo 3.6, sustituyendo su componente BateríaDeServidores *por una instancia de* BateríaDeServidoresTF*:*

instance *csTF : ClienteServidorMúltiple* [*batería : BateríaDeServidoresTF;*]*;*

La compatibilidad con el rol pedir *del cliente está asegurada por la relación de herencia, sin que haya necesidad de volver a comprobar las conexiones entre la batería de servidores y los clientes. Por tanto, hemos obtenido una versión refinada del sistema Cliente/Servidor utilizando servidores no seguros, pero manteniendo las propiedades de la arquitectura original.*

El siguiente ejemplo muestra un segundo refinamiento de la arquitectura Cliente/Servidor.

Ejemplo 3.11 *De forma similar, y siguiendo con nuestro sistema Cliente/Servidor, consideremos ahora la posibilidad de que los servidores puedan fallar mientras están atendiendo una*

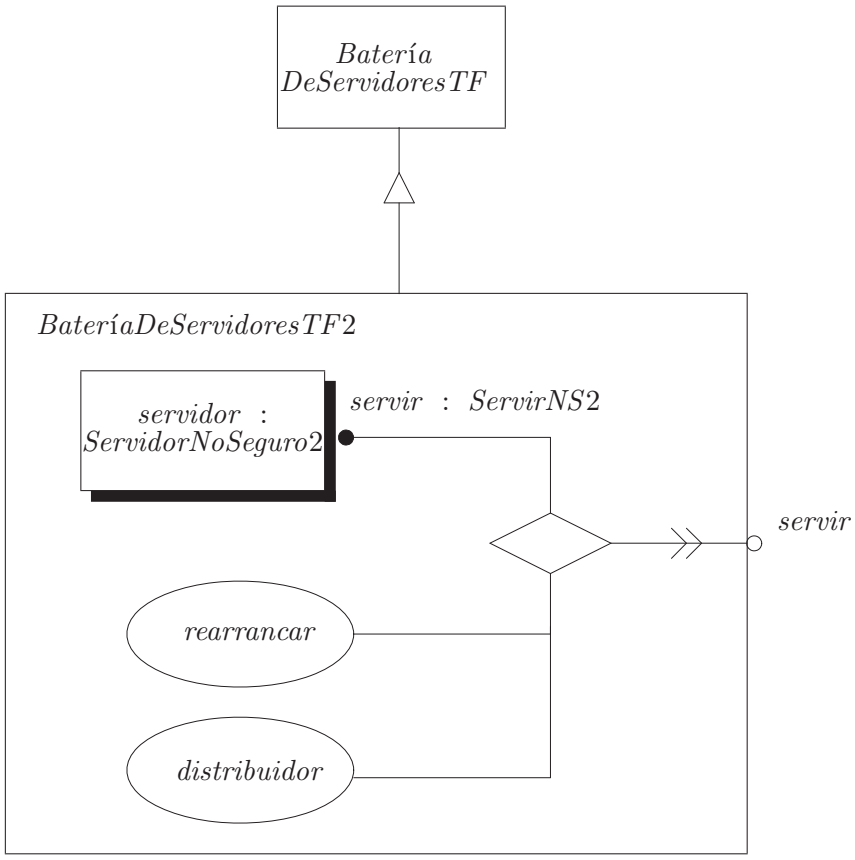


Figura 3.18: Segunda versión de la batería de servidores tolerantes a fallos.

petición del cliente, y no sólo entre peticiones, como era el caso del Ejemplo 3.11. El componente ServidorNoSeguro2*, que figura a continuación, especifica este comportamiento en su rol* servir*.*

```
component ServidorNoSeguro2 {
  interface
    servir : ServirNS2(petición,caída) {
      spec is
        t.petición?(respuesta,ok,err).
          ( t.(new sirviente)respuesta!(sirviente).
              ok!().ServirNS2(petición,caída)
          + t.err!().0 )
      + t.caída!().0;
      }
  composition
    sirviente[ ] : any;
}
```

Estamos ahora en una situación más compleja, en la que no sólo debemos de rearrancar el servidor, sino que tenemos que transmitir de nuevo la petición que no llegó a ser atendida

(incluyendo el canal de respuesta proporcionado originalmente por el cliente). Esta tarea puede ser llevada a cabo haciendo uso de otro adaptador, que llamaremos Distribuidor*, tal como se muestra a continuación:*

```
component BateríaDeServidoresTF2
   extends BateríaDeServidoresTF {
 composition
   servidor[ ] : ServidorNoSeguro2;
   distribuidor : Distribuidor(petición,servidores) {
    spec is
      petición?(respuesta).(ok,err)servidores!(respuesta,ok,err).
        ( Sirviendo(servidores,respuesta,ok,err)
        | Distribuidor(petición,servidores) )
    agent Sirviendo(servidores,respuesta,ok,err) is
      err?() ( new servidor ) servidores!(respuesta,ok,err).
   + ok?().0;
    }
 attachments
   servidor[*].servir(petición,caída),
   distribuidor(petición,servidores),
   rearrancar(caída)     >>     servir(petición);
}
```

Como se puede deducir de la especificación, el distribuidor es quien recibe ahora las peticiones de los clientes, las retransmite a la batería de servidores y crea un proceso Sirviendo *que detecta las caídas de los servidores (simbolizadas por el evento* err*) o la correcta resolución de la petición (simbolizada por el evento* ok*). En caso de error, se arranca de nuevo el servidor, y la petición es retransmitida a la batería de servidores para que sea atendida (indicando el canal de respuesta original por el que se ha de enviar al cliente la referencia al sirviente).*

La interfaz formada por el la combinación del rol servir *de cada uno de los componentes* ServidorNoSeguro2 *y los adaptadores* rearrancar *y* distribuir *se deriva de la interfaz de la batería del servidores del Ejemplo 3.6, como puede ser comprobado mediante la relación de herencia, con lo que el compuesto* BateríaDeServidoresTF2 *es una extensión de* BateríaDeServidores *y su interfaz es compatible con la de los clientes, tal como fueron descritos en el Ejemplo 3.3. La representación gráfica de este sistema puede verse en el Figura 3.18.*

Una vez más, podemos utilizar el refinamiento de arquitecturas con objeto de obtener una arquitectura Cliente/Servidor refinada en la que utilicemos la nueva versión de la batería de servidores:

```
instance csTF2 : ClienteServidorMúltiple [ batería : BateríaDeServidoresTF2; ];
```

3.9 Interpretación de LEDA en el cálculo π

En esta sección presentamos un esbozo de la semántica denotacional de LEDA utilizando como dominio semántico el cálculo π, para lo que mostramos cómo se realiza la interpretación de las especificaciones escritas en LEDA en este álgebra de procesos. De esta forma logramos además que estas especificaciones puedan ejecutarse y analizarse utilizando las herramientas existentes para el cálculo π, como la ya citada MWB [Victor, 1994].

3.9.1 Roles

Comenzaremos por mostrar la traducción de las clases de roles. Para cada clase de roles R de una especificación supondremos la existencia de un nombre de enlace r en el cálculo π. Entonces, si *Rol* es la declaración de dicha clase de roles:

$$
\begin{array}{l}
\mathit{Rol} \equiv \textbf{role}\ R(\mathit{nombres})\ \{ \\
\qquad \textbf{constant}\ \mathit{DeclaraciónDeContantes;} \\
\qquad \textbf{var}\ \mathit{DeclaraciónDeVariables;} \\
\qquad \textbf{spec is}\ \mathit{Especificación}_1\mathit{;} \\
\qquad \textbf{agent}\ R_2\ \textbf{is}\ \mathit{Especificación}_2\mathit{;} \\
\qquad \ldots \\
\qquad \textbf{agent}\ R_q\ \textbf{is}\ \mathit{Especificación}_q\mathit{;} \\
\qquad \}
\end{array}
$$

se tiene que:

$$
\begin{array}{l}
[\![\, \mathit{Rol} \,]\!](r) \stackrel{\text{def}}{=} \textbf{!}\ (\ r?(\mathit{att}).\ \mathit{att}?(\mathit{nombres}).\ (\mathit{consts}, \mathit{vars})\ R_1(\mathit{nombres}, \mathit{consts}, \mathit{vars}) \\
\qquad\qquad \mid\ [\![\, \mathit{DeclaraciónDeConstantes} \,]\!](\mathit{consts}) \\
\qquad\qquad \mid\ [\![\, \mathit{DeclaraciónDeVariables} \,]\!](\mathit{vars})\ )
\end{array}
$$

donde el uso del operador de replicación (**!**) permite crear tantas instancias de la clase de roles como sea necesario. Cada instancia de rol recibe sus nombres libres a través del enlace *att* que lo conecta a otro rol del sistema, como veremos posteriormente.

Además, al crear una instancia de rol se crean nombres de enlace privados para representar a las constantes $\mathit{consts} = (k_1, \ldots, k_{n_k})$ y variables $\mathit{vars} = (v_1, \ldots, v_{n_v})$ que figuran en la declaración de la clase. Estos enlaces, junto con los nombres libres del rol recibidos de la conexión, se transmiten al agente que especifica el estado inicial del rol (representado en este caso como R_1).

Respecto a la declaración de las constantes $k_1, \ldots, k_{n_k}$, se tiene que:

$$
[\![\, \mathit{DeclaraciónDeContantes} \,]\!](k_1, \ldots, k_{n_k}) \stackrel{\text{def}}{=} [\![\, \mathit{Constante} \,]\!](k_1)\ \mid\ \cdots\ \mid [\![\, \mathit{Constante} \,]\!](k_{n_k})
$$

donde

$$
[\![\, \mathit{Constante} \,]\!](k) \stackrel{\text{def}}{=} (\mathit{valor})\ \ \textbf{!}\ (\ k!(\mathit{valor}).\mathbf{0}\ )
$$

mientras que para la declaración de las variables $v_1, \ldots, v_{n_v}$ se tiene que:

$$
[\![\, \mathit{DeclaraciónDeVariables} \,]\!](v_1, \ldots, v_{n_v}) \stackrel{\text{def}}{=} [\![\, \mathit{Variable} \,]\!](v_1)\ \mid\ \cdots\ \mid [\![\, \mathit{Variable} \,]\!](v_{n_v})
$$

donde cada variable se interpreta como un agente:

$$
[\![\, \mathit{Variable} \,]\!](v) \stackrel{\text{def}}{=} (i)\, \mathit{Variable}(v, i)
$$

que se define como:

$$
\mathit{Variable}(v, i) \stackrel{\text{def}}{=}\ v!(i).\mathit{Variable}(v, i)\ +\ v?(n).\mathit{Variable}(v, n)
$$

y en el que i representa el valor inicial de la variable asignado por defecto o establecido en su declaración.

Por otro lado, por cada uno de los agentes R_i que aparecen en la clase de rol, se define un agente en el cálculo π:

$$R_i(nombres, consts, vars) \equiv [\![\ Especificación_i\ ]\!]$$

La especificación de los agentes está escrita en el cálculo π, por lo que la interpretación de *Especificación*$_i$ es directa, si exceptuamos la creación dinámica de componentes con **new** y la difusión de mensajes. La creación de componentes se aborda más adelante, al describir la interpretación de las clases de componentes. Respecto a la difusión de un mensaje a un conjunto de n agentes lectores a través de un enlace múltiple compartido —por ejemplo, $a[*]!(x)$—, ésta se interpreta de la siguiente forma:

$$\begin{aligned}[\![\ a[*]!(x)\ ]\!] \stackrel{\text{def}}{=}\ & (ack)\ (l_1)\ a!(l_1).\ l_1!(x).\ l_1!(ack).\\ & \cdots\\ & (l_n)\ a!(l_n).\ l_n!(x).\ l_n!(ack).\\ & ack?().\ \cdots\ ack?() \qquad\qquad (n\ veces)\end{aligned}$$

Esta implementación de la acción de difusión asegura que ni el agente escritor ni los lectores procedan antes de que el enlace x, objeto de la difusión, haya sido recibido por todos los lectores. Como contrapunto, la recepción complementaria a la difusión se interpreta como:

$$[\![\ a[\ ]?(y)\ ]\!] \stackrel{\text{def}}{=}\ a?(l).\ l?(y).\ l?(ack).\ ack!()$$

Con respecto a la derivación de clases de roles mediante extensión, las clases derivadas se traducen al cálculo π en dos etapas. En primer lugar, se obtienen versiones *aplanadas* de estas clases utilizando para ello las clases progenitoras y reemplazando los agentes redefinidos por versiones actualizadas de los mismos, y añadiendo los nuevos agentes que especifique la clase derivada. A continuación estas especificaciones aplanadas se traducen al cálculo π tal como hemos descrito para las clases en las que no se utiliza herencia.

3.9.2 Componentes

De forma similar a como sucede con los roles, para cada clase de componentes C suponemos la existencia de un nombre de enlace c. Entonces, si *Componente* es la declaración de una clase de componentes:

Componente ≡ **component** C {
constant *DeclaraciónDeConstantes;*
var *DeclaraciónDeVariables;*
interface *DeclaraciónDeRoles;*
composition *DeclaraciónDeSubcomponentes;*
attachments *DeclaraciónDeConexiones;*
}

se tiene que:

$$
\begin{aligned}
[\![\, \mathit{Componente} \,]\!](c) \stackrel{\text{def}}{=} \; & \mathbf{!}\; (\; c?(w).(\mathit{consts}, \mathit{vars}, \mathit{roles}, \mathit{comps}) \\
& \qquad (\; \mathbf{!}\; (\; w!(\mathit{consts}, \mathit{vars}, \mathit{roles}, \mathit{comps}).\mathbf{0}\;) \\
& \qquad \mid [\![\, \mathit{DeclaraciónDeConstantes} \,]\!](\mathit{const}) \\
& \qquad \mid [\![\, \mathit{DeclaraciónDeVariables} \,]\!](\mathit{vars}) \\
& \qquad \mid [\![\, \mathit{DeclaraciónDeRoles} \,]\!](\mathit{roles}, \mathit{rclases}) \\
& \qquad \mid [\![\, \mathit{DeclaraciónDeSubcomponentes} \,]\!](\mathit{comps}, \mathit{cclases}) \\
& \qquad \mid [\![\, \mathit{DeclaraciónDeConexiones} \,]\!](\mathit{comps})\;)\;)
\end{aligned}
$$

donde los nombres $\mathit{consts} = (k_1, \ldots, k_{n_k})$, $\mathit{vars} = (v_1, \ldots, v_{n_v})$, $\mathit{roles} = (\mathit{rol}_1, \ldots, \mathit{rol}_{n_r})$ y $\mathit{comps} = (\mathit{comp}_1, \ldots, \mathit{comp}_{n_c})$ designan enlaces privados que utilizaremos para hacer referencia a las variables, roles y subcomponentes de cada instancia w de la clase, mientras que los nombres $\mathit{rclases} = (r_1, \ldots, r_{n_r})$ y $\mathit{cclases} = (c_1, \ldots, c_{n_c})$ son los enlaces asociados a las clases de los roles $\mathit{rol}_1, \ldots, \mathit{rol}_{n_r}$ y subcomponentes $\mathit{comp}_1, \ldots, \mathit{comp}_{n_c}$, respectivamente.

Cada vez que se crea una instancia de la clase C al recibir un mensaje *c?(w)*, los nombres de las contantes, variables, roles y subcomponentes de la nueva instancia se exportan mediante $w!(\mathit{consts}, \mathit{vars}, \mathit{roles}, \mathit{comps})$ con objeto de utilizarlos, por ejemplo, en las conexiones que se establezcan entre w y otros componentes del sistema.

La creación de una instancia *comp* de una clase de componentes C se lleva a cabo por medio del envío de un nombre nuevo w a través del enlace c asociado a la clase C. A continuación, el nombre w es comunicado a cualquier conexión en la que participe *comp*:

$$
\mathit{Crear}(\mathit{comp}, c) \stackrel{\text{def}}{=} (w)c!(w).\;\; \mathbf{!}\; (\; \mathit{comp}!(w).\;\mathbf{0}\;)
$$

Cuando se instancia una clase de componentes, deben crearse instancias de sus roles y subcomponentes (lo que a su vez instanciará los roles y subcomponentes de estos últimos).

La interpretación de las declaraciones de constantes y variables en el componente se realiza de la misma manera que en los roles. Respecto a la declaración de la interfaz del componente, si $\mathit{DeclaraciónDeRoles} \equiv \; \mathit{rol}_1 : R_1;\; \ldots;\; \mathit{rol}_{n_r} : R_{n_r};$ entonces:

$$
\begin{aligned}
[\![\, \mathit{DeclaraciónDeRoles} \,]\!](\mathit{rol}_1, \ldots, \mathit{rol}_{n_r}, r_1, \ldots, r_{n_r}) \stackrel{\text{def}}{=} \\
[\![\, \mathit{rol}_1 : R_1 \,]\!](\mathit{rol}_1, r_1) \;\mid\; \cdots \;\mid \\
[\![\, \mathit{rol}_{n_r} : R_{n_r} \,]\!](\mathit{rol}_{n_r}, r_{n_r})
\end{aligned}
$$

y para cada $\mathit{rol} \in \mathit{roles}$:

$$
[\![\, \mathit{rol{:}R(n)} \,]\!](\mathit{rol}, r) \stackrel{\text{def}}{=} \mathit{rol}\,?(\mathit{att}).r!(\mathit{att}).\mathbf{0}
$$

Por tanto, cada declaración de rol en la interfaz del componente crea una nueva instancia de la clase correspondiente. Los nombres de la instancia serán establecidos por la conexión *att* que relaciona dicho rol con algún otro rol dentro del sistema.

Cuando la especificación de una clase de roles se declara de forma inmersa en la interfaz de un componente, dicha declaración se expande en una clase separada antes de aplicar la interpretación que acabamos de describir.

Por otro lado, si $\mathit{DeclaraciónDeSubcomponentes} \equiv \; \mathit{comp}_1 : C_1;\; \ldots;\; \mathit{comp}_{n_c} : C_{n_c};$ entonces

$$
\begin{aligned}
[\![\, \mathit{DeclaraciónDeSubcomponentes} \,]\!](\mathit{comp}_1, \ldots, \mathit{comp}_{n_c}, c_1, \ldots, c_{n_c}) \stackrel{\text{def}}{=} \\
[\![\, \mathit{comp}_1 : C_1 \,]\!](\mathit{comp}_1, c_1) \;\mid\; \cdots \;\mid \\
[\![\, \mathit{comp}_{n_c} : C_{n_c} \,]\!](\mathit{comp}_{n_c}, c_{n_c})
\end{aligned}
$$

y para cada $comp \in comps$ tenemos que, o bien:

$$[\![\ comp{:}C\ ]\!](comp, c) \stackrel{\text{def}}{=} Crear(comp, c)$$

si se trata de un componente simple, o bien:

$$[\![\ comp[\ ]{:}C\ ]\!](comp, c) \stackrel{\text{def}}{=} \mathbf{0}$$

si se trata de un vector o colección de componentes. Como vemos, cada declaración de componentes simples crea una instancia w de la clase de componentes correspondiente. Por el contrario, la traducción de un vector de componentes como $comp[\ ]$ es vacía, ya que las instancias del vector se crearán de forma dinámica mediante **new**. La semántica de **new** es la siguiente:

$$[\![\ \mathbf{new}\ comp\ ]\!](c) \stackrel{\text{def}}{=} Crear(comp, c)$$

donde *comp* está declarado como un vector de componentes de clase C.

Por último, y al igual que sucede con los roles, las clases derivadas de componentes se traducen al cálculo π en dos etapas. En una primera fase, se generan versiones aplanadas de estas clases, utilizando para ello la especificación de sus progenitores, y reemplazando los componentes, roles o agentes redefinidos por las versiones actualizadas de los mismos, y añadiendo nuevos roles y agentes según indique la clase derivada. A continuación las especificaciones aplanadas se traducen al cálculo π tal como hemos descrito para las clases en las que no se utiliza herencia.

3.9.3 Conexiones

La sección de conexiones de un componente indica qué roles del sistema están conectados entre sí. Si $DeclaraciónDeConexiones \equiv\ att_1;\ \ldots;\ att_p;$ entonces

$$[\![\ DeclaraciónDeConexiones\ ]\!](comps) \stackrel{\text{def}}{=} \begin{array}{l} [\![\ att_1\ ]\!](comp_{1_i}, comp_{1_d})\ \mid\ \cdots\ \mid \\ [\![\ att_p\ ]\!](comp_{p_i}, comp_{p_d}) \end{array}$$

donde $comp_{1_i}$, $comp_{1_d}$, ..., $comp_{p_i}$, $comp_{p_d}$ son los componentes que aparecen en los términos izquierdo y derecho de cada una de las conexiones. Todos ellos serán subcomponentes de la clase de componentes en la que aparezca la conexión, declarados en su sección de composición, por lo que sus nombres estarán incluidos en *comps*.

En un principio supondremos que las conexiones son binarias, es decir, que conectan pares de roles, para mostrar a continuación conexiones más complejas. En cada conexión, los roles deben compartir los nombres libres (aunque puede haber variaciones en el orden de éstos). Si:

$$att \equiv\ comp_i.role_i(nombres) <> comp_d.role_d(nombres)$$

entonces

$$\begin{array}{l} [\![\ comp_i.role_i(nombres) <> comp_d.role_d(nombres)\ ]\!](comp_i, comp_d) \stackrel{\text{def}}{=} \\ \qquad (izquierdo, derecho, nombres) \\ \qquad (\ \mathbf{!}\ (\ comp_i?(w_i).w_i?(nombres_{w_i}).rol_a!(izquierdo).\mathbf{0}\ ) \\ \qquad \mid\ \mathbf{!}\ (\ comp_d?(w_d).w_d?(nombres_{w_d}).rol_b!(derecho).\mathbf{0}\ ) \\ \qquad \mid\ [\![\ Conexión\ ]\!](izquierdo, derecho, nombres)\ ) \end{array}$$

La acción de entrada $w_i\,?(nombres_{w_i})$ obtiene los nombres privados de los roles y componentes de w_i, los cuales se usan para conectar el rol mediante $rol_a!(izquierdo)$, donde $rol_a \in nombres_{w_i}$ es el nombre privado de una cierta instancia de componente w_i. Obsérvese que la replicación se utiliza aquí para conectar los roles de todas las instancias de $comp_i$, en caso de que éste sea un vector ilimitado de componentes. Lo mismo se aplica para el término derecho de la conexión.

La traducción de las conexiones que acabamos de mostrar se puede extender fácilmente a situaciones en las que se conecten roles de subcomponentes internos, mediante el recorrido del camino hasta llegar a dichos subcomponentes. Así, $comp_1.comp_2.\cdots.rol(nombres)$ se traduciría como:

$$comp_1\,?(w_1).w_1\,?(nombres_{w_1}).nombres_{1_{comp2}}\,?(w_2).w_2\,?(nombres_{w_2})\cdots.rol_i\,!(att).\mathbf{0}$$

donde $nombres_{1_{comp2}} \in nombres_{w_1}$ y $rol_i \in nombres_{w_n}$.

Las conexiones múltiples deben enviar sus *nombres* cada vez que se crea una nueva instancia de los vectores de componentes involucrados. Dependiendo de la notación utilizada en la conexión ($comp[*]$, o $comp[\,]$), entonces bien la *Conexión* es *Compartida* o bien es *Privada*, que se definen respectivamente como:

$$\begin{aligned}[\![\, Compartida \,]\!](izquierdo, derecho, nombres) \equiv\ & \mathbf{!}\ (\ izquierdo!(nombres).\mathbf{0}\ )\ \mid \\ & \mathbf{!}\ (\ derecho!(nombres).\mathbf{0}\ )\end{aligned}$$

$$\begin{aligned}[\![\, Privada \,]\!](izquierdo, derecho) \equiv\ & \mathbf{!}\ (\ (nombres)\ (\ izquierdo!(nombres).\mathbf{0}\ \mid \\ & \qquad derecho!(nombres).\mathbf{0})\ )\end{aligned}$$

Las conexiones privadas crean una copia fresca para cada par de roles de los *nombres* utilizados en la conexión, mientras que en las conexiones compartidas se hace uso siempre de la misma copia. Con respecto a las conexiones estáticas, dichos nombres se utilizan únicamente una vez, por lo que su interpretación es similar a la de las conexiones compartidas, aunque no es necesaria la replicación:

$$\begin{aligned}[\![\, Estática \,]\!](izquierdo, derecho, nombres) \equiv\ & izquierdo!(nombres).\mathbf{0}\ \mid \\ & derecho!(nombres).\mathbf{0}\end{aligned}$$

Aunque hasta ahora nos hemos referido únicamente a conexiones binarias, las conexiones en LEDA pueden involucrar a más de dos roles. De hecho, este es el caso de las conexiones reconfigurables, que involucran roles de tres o más componentes (digamos *izquierdo*, $derecho_1$ y $derecho_2$). Suponiendo definidas las constantes TRUE y FALSE, se tiene que:

$$\begin{aligned}[\![\, Reconfigurable \,]\!](cond, izquierdo, derecho_1, derecho_2) \equiv & \\ \mathbf{!}\ (\ cond\,?(valor).(nombres)\ (\ & izquierdo!(nombres).\mathbf{0}\ \mid \\ & (\ [valor = \textsc{True}]derecho_1!(nombres).\mathbf{0} \\ & +[valor = \textsc{False}]derecho_2!(nombres).\mathbf{0}\ )\ )\ )\end{aligned}$$

Las generalización de la interpretación de las conexiones estáticas y múltiples para tres o más roles se realiza de forma similar.

Por último, la traducción de la especificación completa de una arquitectura consiste en sus clases de roles y componentes en paralelo. Si:

$$Arquitectura \equiv \ Rol_1 \ldots Rol_m \quad Componente_1 \ldots Componente_n \quad \textbf{instance}\ sistema : C_i;$$

entonces

$$\begin{aligned}[\![\, Arquitectura \,]\!] \stackrel{\text{def}}{=} (r_1, \ldots, r_m, c_1, \ldots, c_n) \\ (\ [\![\, Rol_1 \,]\!](r_1) \ \mid \ \cdots \ \mid [\![\, Rol_n \,]\!](r_n) \\ \mid \ [\![\, Componente_1 \,]\!](c_1) \ \mid \ \cdots \ \mid [\![\, Componente_m \,]\!](c_m) \\ \mid \ (sistema)c_i!(sistema).\mathbf{0} \ )\end{aligned}$$

donde la arquitectura se instancia como un componente de la clase C_i mediante $c_i!(sistema).\mathbf{0}$

De nuevo, en caso de que durante la instanciación se realice el refinamiento de la arquitectura, consistente en la extensión implícita de la clase instanciada por medio de la sustitución de algunos de sus componentes por otros derivados de ellos, la generación de la instancia se realiza mediante el aplanamiento previo de la clase extendida de forma implícita.

3.10 Estudio de caso: un sistema de subastas

Como resumen de lo presentado hasta ahora, mostraremos la aplicabilidad de nuestro lenguaje por medio de un ejemplo de un sistema distribuido de subastas, similar al descrito en [Yellin y Strom, 1997], un ejemplo típico de sistema para el comercio electrónico desarrollado sobre Internet.

3.10.1 Descripción

El sistema de subastas vende un conjunto de artículos al mejor postor. Las subastas son moderadas por un subastador y en ellas participan una serie de postores. Al comienzo de cada subasta, el subastador informa a los postores acerca de las características del artículo que se va a subastar, así como de su precio de salida. Durante el transcurso de la subasta, los postores pueden entrar y salir del sistema, lo que es comunicado al subastador.

El subastador recibe pujas de los postores, las cuales son aceptadas o rechazadas dependiendo del precio ofertado. Tanto la aceptación como el rechazo son comunicadas al postor que realizó la puja. Si una puja es aceptada, se informa al resto de los postores del nuevo precio alcanzado por el artículo.

Cuando un postor recibe una actualización del precio del artículo, puede reaccionar bien realizando una nueva puja, en la que oferta un precio más alto, o bien abandonando la subasta. Una vez que un postor ha respondido a una actualización de precio, no debe volver a pujar hasta que reciba un nuevo precio.

La subasta finaliza adjudicando el artículo a la puja más alta. El subastador debe informar de la adjudicación al conjunto de los postores y enviar el artículo al ganador de la subasta. A continuación, el subastador procede a subastar otro artículo.

3.10.2 Especificación del sistema

El sistema descrito puede ser construido a partir de un componente *Subastador* conectado a varios componentes *Postor*, tal como indica la Figura 3.19. Con objeto de obtener un diseño modular, la interfaz del *Subastador* se ha dividido en dos roles: *conectar* modela la conexión

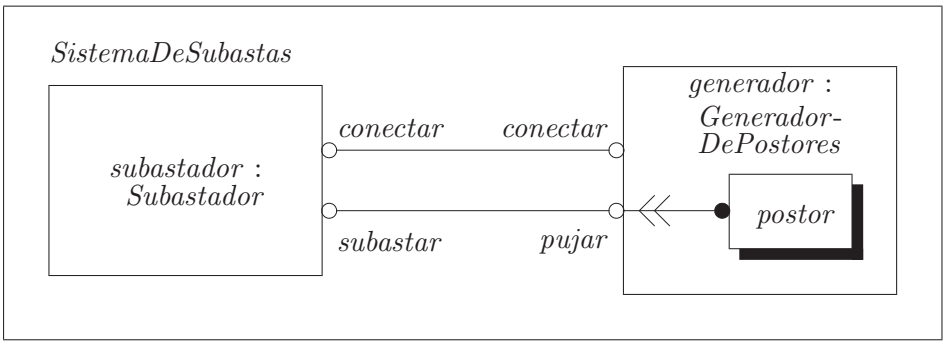


Figura 3.19: Arquitectura del sistema de subastas.

de participantes en cualquier sistema de este tipo, por lo que puede ser fácilmente reutilizado, mientras que el rol *subastar* se refiere al comportamiento específico del componente *Subastador*.

La conexión dinámica de postores es simulada por el componente *GeneradorDePostores*, que genera nuevos componentes *Postor* y los conecta al *Subastador*. El comportamiento de los postores se especifica por medio del rol *pujar*. Tal como se muestra en la figura, el *Generador-DePostores* exporta los roles *pujar* de los postores a través de su propio rol *pujar*, compartiendo todos los postores los nombres de los enlaces. Este rol *pujar* del *GeneradorDePostores* se conecta al rol *subastar* del *Subastador*, de forma que este último pueda difundir los eventos referidos al precio y venta de los artículos a todos los postores.

A continuación mostramos la especificación de estos componentes:

```
component SistemaDeSubastas {
  interface none;
  composition
    subastador : Subastador;
    generador : GeneradorDePostores;
  attachments
    generador.conectar(conexión) <> subastador.conectar(conexión);
    generador.pujar(precio,puja,acuse,vendido,ítem,desconexión) <>
        subastador.subastar(precio,puja,acuse,vendido,ítem,desconexión);
  }

component Subastador {
  var max : Integer := 0;
  interface
    conectar : Conectar(conexión) {
      spec is
        conexión?(). max++. Conectar(conexión);
      }
    subastar : Subastar;
  }
```

La variable *max* del *Subastador* va a indicar el número de participantes conectados en cada momento al sistema. Esta variable relaciona entre sí los dos roles del *Subastador*, motivo por el cual está definida al nivel de este componente.

```
component GeneradorDePostores {
  interface
    conectar : Conectar(conexión) {
      spec is
        conexión!().( new Postor | Conectar(conexión) );
      }
    pujar {
      spec is implicit;
      }
  composition
    postor[ ] : Postor;
  attachments
    postor[*].pujar >> pujar;
  }

component Postor {
  interface
    pujar : Pujar;
  }
```

3.10.3 Especificación del comportamiento

Los roles mencionados describen el comportamiento que deben presentar los componentes del sistema. Su especificación es la siguiente:

```
role Subastar(precio,puja,acuse,vendido,ítem,desc) {
  var valor : Integer;
  spec is
      valor := INICIAL. precio[*]!(valor).
        Subastando(precio,puja,acuse,vendido,ítem,desc);

  agent Subastando(precio,puja,acuse,vendido,ítem) is
      t.precio[*]!(valor). Subastando(precio,puja,acuse,vendido,ítem,desc)
  +   puja?(nuevovalor,respuesta).
        (   t.respuesta!(ACEPTAR). valor := nuevovalor.
              (  t.vendido[*]!(). (artículo)ítem(artículo).
                   Subastar(precio,puja,acuse,vendido,ítem,desc)
              + t.precio[*]!(valor). Subastando(precio,puja,acuse,vendido,ítem,desc) )
        +   t.respuesta!(RECHAZAR). Subastando(precio,puja,acuse,vendido,ítem,desc) )
  +   acuse?(). Subastando(precio,puja,acuse,vendido,ítem,desc)
  +   desc?(). max--. Subastando(precio,puja,acuse,vendido,ítem,desc);
  }
```

El rol *Subastar* describe cómo el *Subastador* controla el desarrollo de la subasta. En primer lugar, el *Subastador* notifica a los *Postores* el precio inicial del artículo, realizando para ello la

difusión de un mensaje a través del canal de precio (*precio*[∗]!(*valor*)). A continuación, el *Subastador* espera por pujas (*puja*?(*nuevovalor*, *respuesta*)); acuses de recibo (*acuse*?()), mediante los cuales los postores indican que renuncian por el momento a pujar, aunque permanecen en la subasta; o desconexiones (*desc*?()), enviadas por los postores. En caso de recibir una puja, el *Subastador* debe decidir su aceptación o rechazo, lo que es notificado al *Postor* que la realizó, utilizando para ello el enlace *respuesta* recibido. Obsérvese como, de nuevo, utilizamos la movilidad para lograr acceso a un canal de respuesta privado. En caso de que se acepte la puja, el *Subastador* puede optar bien por continuar la subasta, para lo que notifica el nuevo precio del artículo a todos los postores, difundiendo de nuevo un mensaje *precio*, o bien por adjudicar el artículo, dando por finalizada su subasta. En este caso, la venta es notificada por medio de la difusión de un mensaje *vendido*, y el artículo es enviado al postor ganador mediante la acción *ítem!*(*artículo*).

role *Pujar*(*precio,puja,acuse,vendido,ítem,desc*) {
var *id : Link;*
spec is
precio[]*?*(*valor*). *Decidiendo*(*precio,puja,acuse,vendido,ítem,desc*)
+*vendido*[]*?*(). *Pujar*(*precio,puja,acuse,vendido,ítem,desc*)*;*

agent *Decidiendo*(*precio,puja,acuse,vendido,ítem,desc*) **is**
precio[]*?*(*valor*). *Decidiendo*(*precio,puja,acuse,vendido,ítem,desc*)
+*vendido*[]*?*(). *Pujar*(*precio,puja,acuse,vendido,ítem,desc*)
+(*nuevovalor*)*puja!*(*nuevovalor,id*). *Esperando*(*precio,puja,acuse,vendido,ítem,desc*)
+*desc!*().**0***;*

agent *Esperando*(*precio,puja,acuse,vendido,ítem,desc*) **is**
id?(*resultado*). ([*resultado*=RECHAZAR]
Pujar(*precio,puja,acuse,vendido,ítem,desc*)
+[*resultado*=ACEPTAR]
Ganando(*precio,puja,acuse,vendido,ítem,desc*))*;*

agent *Ganando*(*precio,puja,acuse,vendido,ítem,desc*) **is**
precio[]*?*(*valor*). *Decidiendo*(*precio,puja,acuse,vendido,ítem,desc*)
+*vendido*[]*?*(). *ítem?*(*artículo*). *Pujar*(*precio,puja,acuse,vendido,ítem,desc*)*;*
}

Por su parte, el rol *Pujar* tiene varios estados, representados cada uno de ellos por medio de un agente. Mientras está en el estado inicial, el *Postor* espera por una notificación del precio del artículo. A continuación entra en el estado *Decidiendo* en el que puede optar por abandonar la subasta enviando un mensaje de desconexión (*desc!*()), o bien por realizar una nueva puja por medio de la acción *pujar!*(*nuevovalor,id*), indicando su oferta junto con un enlace privado para recibir la respuesta. Si ha optado por pujar, el *Postor* entra en el estado *Esperando*. Si la puja es rechazada (debido, por ejemplo, a la realización de una puja por un precio superior por parte de otro *Postor* desde la última actualización del precio del artículo), el *Postor* vuelve al estado inicial, a la espera de recibir el último precio del artículo. Sin embargo, si la puja es aceptada, el *Postor* entra en el estado *Ganando*.

Los mensajes *precio*[]?(*valor*) y *vendido*[]? deben ser aceptados en cualquier estado excepto en *Esperando* (por lo que el *Subastador* está obligado a aceptar o rechazar la puja realizada) y, por lo general, ambos llevan al *Postor* a su estado inicial. No obstante, en el estado *Ganando*,

la recepción de un mensaje *vendido* indica que el artículo ha sido adjudicado a dicho *Postor*, que lo obtiene mediante la acción *ítem?*(*artículo*).

Estos roles completan la especificación del sistema de subastas. Si ahora procediésemos a su análisis, por medio de la relación de compatibilidad descrita en el Capítulo 2, podríamos comprobar que todas las conexiones del sistema de subastas ligan roles que son compatibles unos con otros. Eso asegura que el sistema puede ser construido sin temor a fallos debidos a desajustes en el comportamiento de sus componentes.

Una vez especificada la arquitectura del sistema de subasta, podemos obtener una instancia del mismo, a la que llamamos *subasta*:

instance *subasta : SistemaDeSubastas;*

3.10.4 Extensión del sistema

Como extensión del sistema podemos considerar postores específicos que sigan una estrategia de puja determinada. Con la condición de que su comportamiento sea conforme con el descrito en el rol *Pujar*, estos postores pueden participar en el sistema de subastas. Este es el caso del *PostorLimitado* que figura especificado a continuación:

component *PostorLimitado* **extends** *Postor* {
interface
pujar : PujarLimitadamente;
}

role *PujarLimitadamente*(*precio,puja,acuse,vendido,ítem,desc*)
extends *Pujar*(*precio,puja,acuse,vendido,ítem,desc*) {
redefining *Decidiendo*(*precio,puja,acuse,vendido,ítem,desc*) **adding**
acuse!(). *Rehusando*(*precio,puja,acuse,vendido,ítem,desc*);

agent *Rehusando*(*precio,puja,acuse,vendido,ítem,desc*) **is**
precio[]*?*(*valor*). *acuse!*(). *Rehusando*(*precio,puja,acuse,vendido,ítem,desc*)
\+ *vendido*[]*?*(). *PujarLimitadamente*(*precio,puja,acuse,vendido,ítem,desc*);
}

El *PostorLimitado* tiene limitado su presupuesto a una cierta cantidad de dinero. Si el precio del artículo sube demasiado, renuncia a pujar, entrando en el estado *Rehusando* en el que permanece hasta que se produce la venta del artículo en cuestión. Es posible comprobar, realizando un análisis de herencia de comportamiento, que dicho componente extiende el *Postor* original, por lo que podría reemplazar a este último sin afectar a la compatibilidad de las conexiones del sistema.

Por otro lado, si analizamos la especificación del *Subastador*, podremos observar que no garantiza un comportamiento completamente justo. En efecto, vemos como tras recibir una puja, el *Subastador* puede optar por adjudicar un artículo en cualquier momento, sin dar a todos los participantes oportunidad de superar la última oferta.

Podemos desarrollar un *SubastadorJusto* si tenemos en cuenta el número de reacciones a la última actualización del precio de artículo. Para ello utilizaremos una variable n, que es inicializada cada vez que se acepta una nueva puja. Según esto, el artículo sólo puede adjudicarse

cuando todos los postores hayan reaccionado ante la última actualización de precio (es decir, cuando *n*=*max*).

```
component SubastadorJusto extends Subastador {
 interface
  subastar : SubastarJustamente;
 }

role SubastarJustamente(precio,puja,acuse,vendido,ítem)
   extends Subastar(precio,puja,acuse,vendido,ítem) {
 var n : Integer := 0;
 redefining Subastando(precio,puja,acuse,vendido,ítem) as
  t.precio[*]!(valor). n := 0. Subastando(precio,puja,acuse,vendido,ítem,desc)
 +puja?(nuevovalor,respuesta). n++.
    (  t.respuesta!(ACEPTAR). valor := nuevovalor.
       (  [n=max] vendido[*]!(). (artículo)ítem(artículo). n := 0.
            SubastarJustamente(precio,puja,acuse,vendido,ítem,desc)
       + [n<max] precio[*]!(valor). n := 0.
            Subastando(precio,puja,acuse,vendido,ítem,desc) )
    +  t.respuesta!(RECHAZAR). Subastando(precio,puja,acuse,vendido,ítem,desc) )
 +acuse?(). n++. Subastando(precio,puja,acuse,vendido,ítem,desc)
 +desc?(). max--. Subastando(precio,puja,acuse,vendido,ítem,desc);
 }
```

Este *SubastadorJusto* sigue un comportamiento más predecible que el subastador original, como muestra el hecho de que en su especificación hemos sustituido por guardas en función del valor de la variable *n* las transiciones internas que modelaban como local la decisión de adjudicar o no el artículo a la puja recibida. De nuevo puede demostrarse, por medio de la relación de herencia de comportamiento del Capítulo 2, que lo extiende. Obsérvese que la terminación de las subastas está asegurada, ya que los *Postores* deben reaccionar a toda actualización del precio del artículo, bien sea pujando, rehusando pujar o abandonando la subasta.

Por último, podemos obtener una versión refinada de nuestro sistema de subastas, instanciando su arquitectura con los nuevos componentes *SubastadorJusto* y *PostorLimitado*. Ya que ambos heredan de los especificados originariamente en la arquitectura del sistema de subastas, podemos asegurar que el sistema refinado sigue siendo composicional, sin necesidad de analizar de nuevo la compatibilidad de sus conexiones. Una instancia de esta arquitectura refinada se declararía como:

```
instance subastaJustaLimitada : SistemaDeSubastas[
                                  subastador : SubastadorJusto;
                                  generador.postor : PostorLimitado;
                                  ];
```

Con este estudio de caso hemos pretendido mostrar la aplicación del lenguaje LEDA a la especificación de la arquitectura de una aplicación de una cierta complejidad, y hemos visto cómo se describen los componentes y los roles del mismo, cómo se realizan las conexiones entre roles, y cómo es posible utilizar los mecanismos de extensión y refinamiento de roles y componentes previstos en el lenguaje para definir sistemas que incorporen variaciones sobre lo originalmente indicado.

Para acabar con este capítulo dedicado al lenguaje LEDA, realizaremos una comparación de nuestro lenguaje con otras propuestas existentes en este campo. A ello es a lo que se consagra la sección siguiente.

3.11 Comparación con otras propuestas

Prueba del creciente interés que los aspectos arquitectónicos están recibiendo por parte de la comunidad científica es el hecho de que en los últimos años se han propuesto toda una serie de ADLs. Ejemplos de ello son Rapide [Luckham et al., 1995], UniCon [Shaw et al., 1995], Darwin [Magee et al., 1995, Magee y Kramer, 1996], Wright [Allen y Garlan, 1997, Allen et al., 1998], o C2 [Medvidovic et al., 1996], entre otros. Esta gran variedad de propuestas ha llevado incluso al desarrollo de ACME [Garlan et al., 1997], que más que como lenguaje de descripción de arquitecturas propiamente dicho, ha sido presentado como lenguaje de intercambio entre los diferentes ADLs, lo que permitiría la puesta en común de los distintos enfoques seguidos y de las herramientas asociadas a cada uno de ellos. Las características más relevantes de los ADLs citados ya fueron expuestas en la Sección 1.4.5. Aquí realizaremos un estudio comparativo de los mismos frente a nuestra propuesta.

Para comenzar, debemos decir que no obstante el activo trabajo desarrollado en este campo, no existe aún un consenso claro por parte de la comunidad científica respecto a lo que es un ADL, ni tampoco respecto a qué aspectos del software deben ser modelados por este tipo de lenguajes [Medvidovic y Taylor, 2000]. Esto ha dado lugar a una gran diversidad de enfoques entre las distintas propuestas.

Así, si bien en todos los ADLs considerados el concepto de *componente* constituye un elemento fundamental en la estructura del lenguaje, haciendo hincapié de esta forma en los aspectos composicionales del diseño arquitectónico, no ocurre lo mismo con el de *conector*. Para algunos autores existe una clara diferencia entre ambos conceptos, diferencia que debe reflejarse por tanto en los ADLs.

Esta dicotomía componentes/conectores es la base fundamental de lenguajes como UniCon, en el que las abstracciones que representan a los componentes, y muy especialmente a los conectores, están predeterminadas y forman parte del propio lenguaje. Este control estricto sobre los conectores es lo que permite a UniCon la generación de un sistema ejecutable a partir de la especificación arquitectónica. Para ello basta con asociar cada componente del sistema con el elemento de software que lo implementa (ya sea un módulo binario, un fichero, etc.) y conectar estos componentes mediante conectores de los tipos indicados en la arquitectura, utilizando para ello los mecanismos asociados a cada tipo de conector (lenguajes de *script*, instrucciones de compilación y ensamblado, etc.).

Tal como comentamos en la Sección 1.4.5, esta posibilidad de obtener de forma automática el sistema final es la característica más relevante de UniCon, pero el precio que hay que pagar por ello es una excesiva rigidez en el lenguaje. En primer lugar, no es posible la definición de nuevos tipos de componentes o conectores. En segundo lugar, si bien UniCon permite la composición de componentes, no ocurre lo mismo con los conectores; éstos son simples y de los tipos predefinidos.

Por otra parte, UniCon carece de base formal, y deja de lado, por tanto, todos los aspectos de análisis y verificación de propiedades que, desde nuestro punto de vista, son esenciales al menos en las fases iniciales del desarrollo de un sistema de software de cierta envergadura. En nuestra opinión, y como ya hemos comentado, este lenguaje está excesivamente centrado

en la obtención de una implementación ejecutable a partir de la descripción arquitectónica, lo que le hace estar demasiado ligado a aspectos técnicos relacionados con la composición modular utilizando los lenguajes de programación habituales, de manera que las descripciones de interfaz de los componentes se limita a la enumeración de los elementos de interconexión (ficheros, *pipes*, *sockets*, procedimientos o métodos exportados, etc.), que utilizan dichos componentes.

Por el contrario, otras propuestas consideran los conceptos de componente y conector desde un punto de vista más abstracto y más alejado de los aspectos prácticos de la implementación. Éste es el caso del lenguaje Wright, en el que la diferenciación entre el concepto de componente y el de conector sigue siendo radical, llevando incluso a notables diferencias a la hora de especificar la interfaz de uno y otro tipo de elementos. Podemos resumir aquí estas diferencias en el hecho de que la interfaz de los componentes está formada por un conjunto de *puertos* que describen el comportamiento del componente tal como es observado desde el exterior, lo que podríamos llamar —utilizando términos fotográficos— una descripción *en positivo*. En cambio, la interfaz de los conectores viene dada por un conjunto de *roles* que no indican directamente el comportamiento observable del conector, sino —justo al contrario— el comportamiento que deben tener los componentes que deseen conectarse a través de él (con lo que podríamos decir que dan una imagen *en negativo* del conector).

El formalismo utilizado como base de este lenguaje —el álgebra de procesos CSP— permite la especificación del comportamiento de los componentes y conectores en Wright con una riqueza semántica y expresividad muy superiores a las de UniCon, de manera que las descripciones de interfaz de estos elementos no se reducen a signaturas básicamente sintácticas, sino que permiten incorporar aspectos tales como los protocolos de interacción utilizados por dichos elementos. Esta base formal es la que posibilita así mismo el análisis de las arquitecturas especificadas para verificar el cumplimiento de diversas propiedades.

No obstante, es la propia expresividad ofrecida por Wright a la hora de definir nuevos tipos de componentes y conectores la que desvirtúa la diferenciación entre ambos conceptos. Así, a la hora de modelar un determinado elemento de software por medio de este lenguaje, habremos de decidir si lo consideramos como un componente o un conector, diferenciación que es en muchas ocasiones artificiosa. Baste por ejemplo señalar que en Wright un fichero es considerado generalmente como un componente, mientras que una *pipe* es un ejemplo típico de conector utilizado por los autores para ilustrar este concepto, y ello a pesar de que ficheros y *pipes* presenten la misma interfaz y tengan un funcionamiento similar. Más aún, la posibilidad que ofrece este lenguaje de definir componentes y conectores de *grano más grueso*, mediante la composición de otros componentes y conectores, llevaría a desvirtuar totalmente la diferenciación, puesto que, en principio, permitiría la creación de compuestos híbridos, que compartiesen las características de ambas categorías y tuviesen por tanto un interfaz mixta, formada por un conjunto de puertos y roles.

Por los motivos que acabamos de exponer, y también para simplificar todo el aparato formal necesario para el análisis y verificación de propiedades, es por lo que hemos decidido dotar a nuestro lenguaje de un único bloque fundamental: el componente, con una interfaz de comportamiento expresada —*en positivo*— por medio de una serie de roles. No quiere esto decir que no seamos conscientes de la importancia de los conectores, sino simplemente que no les damos un tratamiento específico y diferenciado en nuestra propuesta: todos los elementos de interés arquitectónico se modelan como tipos específicos de componentes.

Existen además otras diferencias significativas entre LEDA y Wright. En primer lugar, tenemos las diferencias resultantes de la elección del formalismo de base, como es la posibilidad de descripción de arquitecturas dinámicas que presenta nuestro lenguaje frente al estatismo de los

sistemas descritos en Wright. Ya hemos discutido estos aspectos ampliamente en la Sección 2.5, por lo que nos limitamos aquí a hacer referencia a ella. Por otro lado, también debemos señalar la inexistencia en Wright de mecanismos de extensión o redefinición de componentes, con lo que no existe tampoco la posibilidad de refinamiento de arquitecturas que, en nuestra propuesta, se derivan de estos mecanismos. Por último, se prescinde totalmente en este lenguaje de los aspectos de implementación, siendo un lenguaje exclusivamente de especificación y verificación formal. Frente a ello, nuestra propuesta contempla la generación de código a partir de las especificaciones arquitectónicas, si bien no con el objetivo de obtener directamente el sistema final, como era el caso de UniCon, si de obtener al menos un prototipo del sistema que pueda ser utilizado, mediante desarrollo incremental y evolutivo, hasta llegar al sistema final, aspectos que abordaremos en el Capítulo 4. En este sentido, podríamos considerar que nuestra propuesta se sitúa en un punto intermedio entre los dos extremos representados por Wright y UniCon, intentando llegar a un equilibrio entre un enfoque teórico y formal, pero a la vez más expresivo y flexible, como es el del primero, y uno más práctico, pero a la vez más estricto, como es el de este último.

Un ADL que podríamos considerar conceptualmente próximo a nuestra propuesta es Darwin. En efecto, Darwin prescinde también de la diferenciación entre componentes y conectores, y utiliza el cálculo π como base formal del lenguaje, al menos a la hora de expresar su semántica, lo que permite dotar a este lenguaje de mecanismos de reconfiguración y conexión dinámicas semejantes a las conexiones múltiples entre colecciones de componentes en LEDA, que se van llevando a cabo a medida que se crean los citados componentes.

La especificación de la interfaz de un componente en Darwin está dividida en dos partes, estando indicada, por un lado, por medio de una serie de servicios que el componente proporciona y, por el otro, por los de servicios que el componente requiere para poder llevar a cabo su labor, a semejanza de las interfaces de exportación y exportación utilizadas en algunos lenguajes modulares de programación. En cierta medida, estos servicios pueden considerarse análogos a los roles utilizados en nuestra propuesta, aunque son, por lo general, de grano más fino, mientras que la distinción entre servicios ofrecidos y requeridos implican en Darwin una serie de restricciones a la hora de interconectar componentes que no es necesario imponer en el caso de LEDA.

No obstante las semejanzas citadas, la propuesta original de Darwin prescinde de la especificación del comportamiento de los componentes, con lo que dicha interfaz se limita a la enumeración sintáctica de los nombres de los servicios requeridos u ofrecidos, sin indicar las restricciones de comportamiento y consideraciones adicionales que el componente asume a la hora de requerir dichos servicios o de ofrecerlos a otros. Como por otro lado, el lenguaje carece de mecanismos de especialización y extensión de componentes, las posibilidades de análisis se limitaban únicamente a la comprobación de tipos en las conexiones entre componentes basadas únicamente en el uso de nombres de servicios (ofrecidos y requeridos) idénticos. En trabajos posteriores, [Magee et al., 1999] ha sido propuesta la especificación del comportamiento de los componentes por medio de sistemas de transiciones etiquetadas (LTS) de estado finito. Sin embargo, el formalismo utilizado en estos últimos trabajos carece de la capacidad de expresar movilidad, lo que ya no le hace adecuado para la descripción y análisis de arquitecturas dinámicas.

Otro ejemplo de los escasos ADLs que permiten la especificación de arquitecturas dinámicas es Rapide, que proporciona mecanismos tanto para la instanciación como para la reconfiguración de las conexiones entre componentes de forma dinámica. Existen diversas semejanzas entre los aspectos 'estructurales' de Rapide y de LEDA, como por ejemplo, el hecho de que ninguno de los dos lenguajes incorpore el concepto de conector como una entidad propia del lenguaje, el que las

interacciones entre componentes se modelen en ambos mediante paso de mensajes, o los referidos a los distintos tipos de conexiones entre componentes. Sin embargo, también existen diferencias significativas entre ambos, como son la ausencia del concepto de rol en Rapide, lo que determina que la interfaz de un componente es monolítica (y por tanto, también lo es la especificación del protocolo asociado a la interfaz), o el hecho de que el formalismo de base utilizado en ambos casos es totalmente diferente. En el caso de Rapide, se trata de un lenguaje basado en eventos, en el que el comportamiento de los componentes se describe mediante los llamados *posets* (*partially ordered sets of events*, conjuntos de eventos parcialmente ordenados), que podríamos considerar como un nivel intermedio entre las trazas de eventos —que representan una única secuencia lineal de eventos—, y los procesos de un álgebra de procesos —que describen todos los eventos que un componente puede presentar. Esta descripción del comportamiento mediante *posets* se utiliza en Rapide para el prototipado de sistemas de software, con objeto de construir patrones de eventos representativos del comportamiento de una determinada aplicación. Por otra parte, Rapide permite el refinamiento y rastreo de protocolos a diferentes niveles de abstracción, de una forma más próxima a la noción de refinamiento de acciones de Aceto y Hennesy al que hacemos referencia en la Sección 2.5, que a nuestra relación de herencia y extensión de protocolos.

Hasta aquí nos hemos centrado en ADLs que —como LEDA— podríamos considerar *de propósito general*, es decir, que no presuponen el uso de un determinado estilo arquitectónico ni están enfocados a un dominio de aplicación en particular. No obstante, existen numerosos ADLs que restringen la forma de las arquitecturas descritas o su dominio de aplicación con objeto aprovechar este mejor conocimiento del sistema o del problema para proporcionar mayores ventajas a la hora de generar un prototipo o la propia aplicación o para realizar análisis de propiedades más específicas. Como ejemplo de este tipo de ADLs, de utilización más restringida, consideraremos el lenguaje C2.

A diferencia del resto de los lenguajes mencionados hasta ahora, las arquitecturas descritas en C2 están organizados en niveles, interconectados mediante un *bus* de datos que se encarga de la transmisión de mensajes, comunicando los componentes de un nivel con los del siguiente.

Aparte de estas particularidades del lenguaje C2, y ya en relación con nuestra propuesta de ADL, cabe señalar que en C2 es posible la adición, supresión y reconexión de componentes de forma dinámica, lo que permite la representación de arquitecturas dinámicas, aunque limitado este dinamismo por los condicionantes del estilo arquitectónico adoptado, de forma que se mantenga la estructuración en niveles de la arquitectura. No existe sin embargo la posibilidad de realizar refinamiento de componentes o arquitecturas, ni se indica de éstos más que su interfaz, es decir, no hay descripción del protocolo seguido por los componentes como era el caso de LEDA, Wright o Rapide. Por otro lado, gracias a las restricciones impuestas por el lenguaje, es posible obtener un sistema directamente ejecutable a partir de una especificación en Rapide, a través de un paso intermedio en lenguaje C.

Entre las diferentes comprobaciones que se pueden realizar sobre una arquitectura descrita en Rapide, está lo que los autores llaman la *conformidad de protocolos*, que va más allá de los métodos individuales ofrecidos y requeridos por los componentes, permitiendo especificar restricciones en el orden en el cual dichos métodos pueden ser invocados. Sin embargo, esta especificación y comprobación del comportamiento de los componentes se lleva a cabo mediante el uso de precondiciones y postcondiciones, lo que en determinadas situaciones resulta sencillo de realizar, pero que en ocasiones puede convertirse en una tarea tediosa que resulte en una especificación complicada y difícil de entender [Buechi y Weck, 1999]. Desde nuestro punto de vista (avalado por su uso práctico en el campo de la especificación de protocolos de comunicaciones) el uso de álgebras de procesos (o de lenguajes derivados de ellos) es la opción más acertada para la descripción comprensible y analizable de patrones de comportamiento.

3.12 Conclusiones

En este capítulo hemos presentado LEDA, un lenguaje formal de descripción de arquitectura basado en el cálculo π y en los conceptos y relaciones formalizados en el Capítulo 2. El lenguaje se articula en dos niveles, el de los componentes y el de los roles.

Los componentes representan partes o módulos del sistema y cada uno de ellos aporta una determinada funcionalidad al mismo. Su inclusión como entidad fundamental dentro del lenguaje permite enfocar el desarrollo de aplicaciones desde un punto de vista composicional, es decir, como un proceso de construcción basado en el ensamblado de elementos que —posiblemente— tienen distinta procedencia.

Por su parte, los roles son abstracciones que describen parcialmente la interfaz de comportamiento de los componentes y se especifican por medio de una notación derivada del cálculo π. Así, un componente estará descrito por varios roles, cada uno de los cuales muestra el comportamiento del componente en relación a otro componente del sistema del que ambos forman parte.

La arquitectura de las aplicaciones descritas en LEDA viene determinada por las *conexiones* que se establecen entre los roles de los distintos componentes del mismo. Al estar estos roles descritos de manera formal, es posible realizar un análisis de las conexiones del sistema, con objeto de averiguar si son compatibles y si la arquitectura de la que forman parte es composicional. Para ello se hace uso de la relación de compatibilidad y de los resultados relativos a dicha relación que fueron presentados en el Capítulo 2.

También hemos expuesto cómo estas conexiones entre roles pueden ser —en nuestro lenguaje— de varios tipos: estáticas, reconfigurables, múltiples y de exportación, lo que permite la descripción de arquitecturas dinámicas, es decir, que el lenguaje pueda utilizarse para representar sistemas cuya estructura varíe en el tiempo. Este es uno de los aspectos en los que nuestra propuesta realiza aportaciones significativas respecto a gran parte de los ADLs tomados como referencia, que sólo permiten la descripción de arquitecturas estáticas e inmutables.

Otro aspecto significativo de nuestra propuesta es la posibilidad de derivar roles unos de otros, de forma que podemos redefinir y extender el comportamiento indicado por un rol, obteniendo a partir de él una versión derivada que mantiene la compatibilidad en las conexiones en las que participaba el rol original. Las nociones de derivación y refinamiento se aplican también a los componentes y a las propias arquitecturas. En el caso de los componentes, el refinamiento puede utilizarse para relacionar descripciones realizadas con distintos niveles de abstracción —unas más generales y otras más detalladas—, dando lugar a un desarrollo evolutivo e incremental de la especificación arquitectónica. En el caso de una arquitectura, el refinamiento consiste en la instanciación de alguno de sus componentes, reemplazándolo por una versión derivada del mismo. Esto da lugar a que podamos considerar cualquier componente de una arquitectura descrita en LEDA como un parámetro formal de la misma, de forma que el lenguaje no describe ya arquitecturas concretas, sino marcos de trabajo composicionales, que pueden ser instanciados y reutilizados tantas veces como sea necesario.

Además de los tres conceptos básicos ya citados: componentes, roles y conexiones, en LEDA existen otros elementos que puede formar parte de una arquitectura: los adaptadores. Estos adaptadores son pequeños elementos de coordinación, descritos por medio del cálculo π, que permiten la conexión de roles que, de otra forma, serían incompatibles.

El uso de todos estos elementos y mecanismos del lenguaje ha sido ilustrado por medio de una serie de ejemplos de creciente complejidad, basados en el patrón arquitectónico Cliente/Servidor, y distribuidos a lo largo de todo el texto. Adicionalmente, hemos desarrollado un caso de

estudio: un sistema distribuido de subasta electrónica, ejemplo típico de aplicación de comercio electrónico desarrollada sobre Internet.

Con este capítulo hemos pretendido presentar, de forma práctica e intuitiva, el lenguaje LEDA, así como esbozar su semántica denotacional por medio de su interpretación en el cálculo π. La descripción detallada de la sintaxis del lenguaje se encuentra en el Apéndice A.

Capítulo 4

El proceso de desarrollo a partir de la especificación en LEDA

La consolidación de los primeros estándares relacionados con los componentes de software ha modificado sustancialmente la forma en la cual se construyen aplicaciones distribuidas y se interconectan sus componentes. Las plataformas de componentes como CORBA, DCOM o JavaBeans se han convertido en una alternativa importante a la hora de construir este tipo de aplicaciones en un plazo corto de tiempo [Szyperski, 1998]. Sin embargo, aunque las plataformas citadas ofrecen mecanismos para construir, interconectar y reutilizar componentes, no abordan aspectos relacionados con la definición, verificación y reutilización de la arquitectura de las aplicaciones [Krieger y Adler, 1998].

La Arquitectura del Software trata de suplir estas carencias haciendo explícita la descripción de la arquitectura de los sistemas. Esta descripción sirve, ya en las fases iniciales del proceso de desarrollo, para fijar y comunicar el conocimiento acerca del sistema, representado con un alto grado de abstracción, de forma que resalten las partes o componentes que lo forman y las relaciones que se establecen entre dichos componentes.

En el capítulo anterior hemos presentado LEDA, un lenguaje para la descripción arquitectónica del software. La base formal del lenguaje permite el análisis de las arquitecturas descritas, para determinar si verifican o no diversas propiedades. Sin embargo, el inconveniente que suelen presentar los ADLs basados en formalismos es que, si bien nos permiten diseñar una arquitectura libre de errores, no suelen llegar a una implementación que conserve dichas propiedades. Normalmente el paso entre la especificación y la implementación sigue siendo responsabilidad del diseñador, lo que conlleva un esfuerzo considerable y sin garantía de éxito, al no estar asegurado que el sistema resultante se comporte igual que la especificación original.

Por estos motivos nuestra propuesta no se limita a la especificación y análisis de la arquitectura. Nuestra intención en este capítulo es avanzar un paso más y proporcionar una guía para el proceso de desarrollo a partir de la descripción arquitectónica que permita garantizar el desarrollo de aplicaciones robustas y escalables. El reto consiste en generar una implementación fiel a la especificación arquitectónica original, que sirva de base para el prototipado y el desarrollo evolutivo e incremental del sistema [Boehm, 1988, Jacobson et al., 1999], obteniéndose así una aplicación robusta y libre de errores. En este capítulo esbozamos un proceso de desarrollo que contempla la generación automática de componentes a partir de su especificación en LEDA.

4.1 Un proceso basado en las interfaces

La aplicación de los principios de la Arquitectura del Software y de la Ingeniería del Software basada en Componentes implica un replanteamiento del proceso de desarrollo de aplicaciones distribuidas. El fin último de ambas disciplinas es facilitar el desarrollo, mantenimiento y evolución de los sistemas de software, basándose para ello en un enfoque composicional, según el cual los componentes definen las restricciones que imponen de cara a colaborar unos con otros. Esto nos lleva a un diseño basado en las interfaces, representación abstracta de los componentes, que se centra en buscar soluciones para un sistema a partir de una arquitectura de componentes.

Este enfoque permite mejorar el proceso de desarrollo, realizando con detenimiento el diseño de las interfaces del sistema. De esta forma, el diseñador podrá centrar su atención en definir las colaboraciones que se vayan a establecer entre los componentes, lo que servirá de base para comprender la arquitectura del sistema y facilitar la reutilización y el reemplazamiento de implementaciones que cumplan con los requisitos especificados por la interfaz.

Existen varias alternativas a la hora de definir el soporte lingüístico para un proceso de desarrollo basado en las interfaces [Bosch, 1997]. Una de ellas consiste en utilizar los lenguajes orientados a objetos (LOO) ya existentes. Este es el caso de las JavaBeans, que son el resultado de extender las interfaces Java dotándolas de la capacidad de introspección. Si bien la ventaja de este enfoque es que se utiliza el propio lenguaje de implementación para definir y componer componentes, presenta igualmente algunas desventajas, heredadas de las limitaciones de la tecnología de objetos [Brown y Wallnau, 1998]. Por ejemplo, en relación a la reemplazabilidad sería necesario poder expresar qué componentes pueden desempeñar un determinado rol dentro de la arquitectura de un sistema. Sin embargo, utilizando un LOO se podrá únicamente definir una interfaz abstracta que representa la signatura de una serie de métodos, pero no los protocolos de interacción que impone una determinada arquitectura.

Otro tipo de propuestas son las llamadas *generativas*, que se basan en la generación de código a partir de un lenguaje o notación de más alto nivel que los lenguajes de implementación. Una propuesta generativa asume, por tanto, que los desarrolladores trabajarán tanto a nivel de especificación, como a nivel de implementación. Para la especificación se utiliza un lenguaje de alto nivel a partir del cual se generará código, y aquellos aspectos no definidos en la especificación se completan durante la fase de implementación. Este enfoque facilita la búsqueda de la arquitectura más adecuada para la aplicación, para posteriormente completarla con detalles de implementación de bajo nivel. A este respecto, una opción muy interesante nos la proporcionan los lenguajes de descripción de arquitecturas, especialmente los que se sustentan en una base formal, puesto que permiten especificar interfaces más completas, donde además de los métodos que exporta se especifique el protocolo de comportamiento del componente.

Nuestra propuesta es generativa, partiendo de LEDA como lenguaje de especificación. Los componentes, los roles que representan la interfaz de dichos componentes, y la arquitectura de un sistema, descrita por medio de las conexiones que relacionan los roles de unos componentes y otros, se describen en primer lugar en este lenguaje.

Una vez realizada la especificación, será necesario proceder a la validación de la arquitectura, por medio del análisis de la compatibilidad de cada una de sus conexiones. También se realiza en este momento la verificación de las relaciones de herencia y extensión de roles y componentes. De esta forma se valida la arquitectura del sistema, lo que nos permite comprobar que la especificación satisface determinadas propiedades. Tal como comentamos en el Capítulo 2, esta validación puede realizarse con ayuda de herramientas automáticas de análisis.

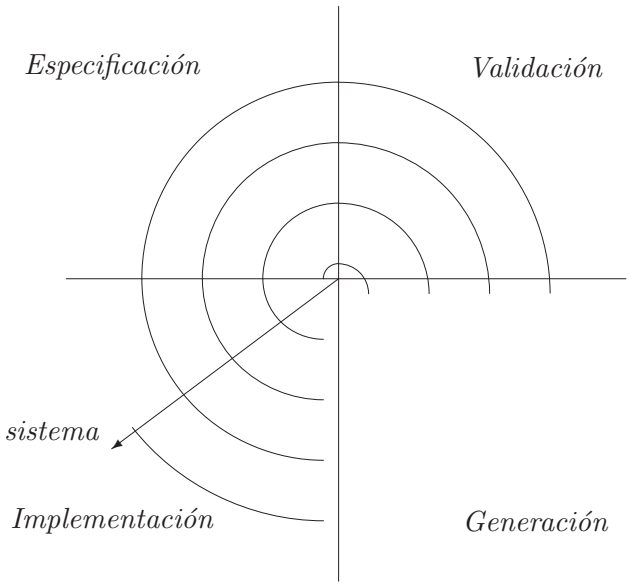


Figura 4.1: Proceso de desarrollo de la especificación a la implementación.

A continuación la especificación será traducida al lenguaje de implementación, conservando sus propiedades, en particular la compatibilidad entre componentes demostrada durante el análisis del sistema. La implementación resultante debe contener el código correspondiente a los protocolos de interacción seguidos en las conexiones entre componentes.

El esquema de generación de código ha sido diseñado teniendo en cuenta los siguientes objetivos:

- El código generado ha de ser un prototipo ejecutable de la aplicación final.
- Debe fomentarse el desarrollo evolutivo e incremental. Inicialmente sólo es necesaria la especificación de los roles de los componentes, dejándose para más adelante la descripción e implementación de las computaciones.
- La implementación estará basada en los patrones de interacción descritos en la especificación, debiendo así mismo conservar todas las propiedades demostradas al respecto durante la fase de verificación.
- El código generado deberá permitir la manipulación directa por parte del diseñador.
- El mecanismo de comunicación utilizado para interconectar componentes debe permanecer oculto al diseñador.

El resultado de la generación de código es un prototipo que servirá como base para la implementación del sistema final, siguiendo un proceso evolutivo e incremental que permite probar el sistema según va siendo construido y en el que los encargados del desarrollo únicamente tendrán que agregar lo referente a la computación sobre los datos intercambiados por los componentes. El lenguaje utilizado para la implementación es Java, aunque nuestra propuesta tiene validez para cualquier lenguaje orientado a objetos.

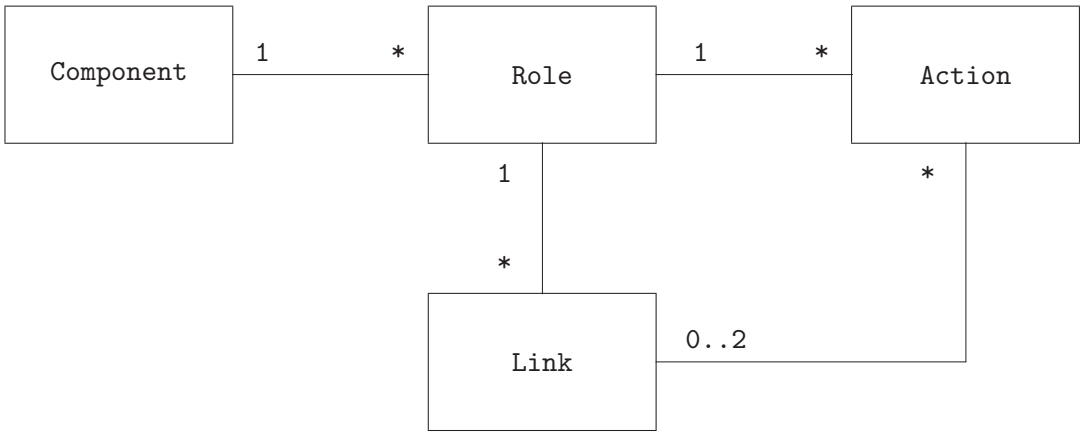


Figura 4.2: Clases fundamentales de la implementación en Java.

La Figura 4.1 representa de forma gráfica este proceso de desarrollo, basado en el ciclo de vida en espiral [Boehm, 1988] y que está dividido en cuatro fases: especificación, validación, generación e implementación, que se repiten de forma cíclica, de forma que el sistema va evolucionando y engrosando desde el prototipo inicial (en el centro) hasta el sistema final.

4.2 Esquema general de traducción

Una especificación en LEDA consta de tres elementos fundamentales: *componentes*, *roles* y conexiones entre roles. Las conexiones se llevan a cabo por medio de la compartición de *enlaces* sobre los cuales los roles realizan *acciones* de lectura y escritura. Por tanto, componentes, roles, enlaces y acciones son las cuatro clases Java alrededor de las cuales se articula la traducción. La Figura 4.2 muestra las relaciones entre estas clases, y en ella se puede apreciar cómo la traducción a Java de una especificación en LEDA se organiza a partir de un conjunto de componentes, representados por la clase `Component`, cada uno de los cuales se asocia a una serie de roles `Role`, que disponen a su vez de una serie de enlaces `Link`. Estos enlaces sirven para que los roles —y por ende los componentes— puedan comunicarse unos con otros. La comunicación se lleva a cabo mediante la realización de acciones `Action` sobre los enlaces.

Una vez determinadas las clases en torno a las que estructurar la generación de código, debemos abordar los aspectos de control. Podemos considerar varias posibilidades, dependiendo de en qué objetos hagamos residir el control del sistema resultante de la traducción:

- El control del sistema reside en los roles, siendo los componentes elementos *pasivos* controlados por éstos.
- El control del sistema reside en los componentes, mientras que los roles simplemente ofrecen métodos para la lectura y escritura de los enlaces correspondientes.
- El control está distribuido entre los componentes y los roles.

Tras considerar detenidamente estas alternativas, se optó por la primera de ellas. Entre las razones que llevaron a esta decisión destaca el que esta opción presenta una clara separación

de los aspectos de computación y coordinación [Aksit et al., 1993], centralizando en los roles la coordinación o control del sistema, mientras que los componentes se encargan de las computaciones. Es evidente que ésta es la opción más natural a partir de una especificación en LEDA, permitiendo así alcanzar más fácilmente los objetivos anteriormente expuestos.

En efecto, cualquiera de las otras dos opciones implica dotar de cierto grado de responsabilidades de control a los componentes, lo que presenta complicaciones puesto que en una especificación en LEDA no se describe el control interno de dichos componentes, sino únicamente su composición y su interfaz. Para resolver estas dificultades habría que realizar una especificación completa de los componentes, pero precisamente nuestro objetivo es evitar la complejidad y los detalles innecesarios asociados a una especificación de este tipo, por lo que nos hemos limitado a describir la interfaz del componente por medio de una serie de roles independientes, de menor complejidad y susceptibles de ser sometidos a análisis local.

Por tanto, el esquema de traducción de la especificación que hemos seguido implica asignar a los roles la funcionalidad de controlar sus respectivos componentes, actuando además de conectores entre ellos, de forma similar a la propuesta en [Ducasse y Richner, 1997]. Por su parte, los componentes encapsularán datos y computación por medio de una serie de métodos que inicialmente estarán vacíos, siendo el protocolo de comunicación descrito en los roles totalmente ajeno a ellos.

Según esto, los componentes simplemente leen o escriben, a través de sus roles, en una serie de enlaces, sin tener conocimiento de quiénes están al otro lado. Esta forma de interconexión difiere de la que se utiliza normalmente en las plataformas de componentes, en las que éstos solicitan a la plataforma conectarse o pedir un servicio a otro componente. La desventaja de este último enfoque es que los componente deben indicar de forma explícita alguna información acerca de los destinatarios de sus mensajes. En nuestro caso, los componentes no tienen información sobre el resto del sistema. De forma similar, los roles no conocen las computaciones que se realizan con los datos transmitidos a través de los enlaces, limitándose a invocar los métodos de los componentes correspondientes a las acciones de escritura y lectura. Componentes y roles están por tanto ligados muy débilmente entre sí, lo que facilita su reutilización de forma separada.

El prototipo inicial estará formado básicamente por los roles, que invocarán los métodos de los componentes. El desarrollo incremental del sistema se llevará a cabo implementando dichos métodos, de forma que se consiga dar al prototipo la funcionalidad adecuada.

Ejemplo 4.1 *La Figura 4.3 muestra de forma más detallada la estructura de la solución propuesta. En ella se representa (prescindiendo de momento de las acciones) el diagrama de instancias de un sistema que se corresponde con la siguiente especificación en LEDA:*

```
component Sistema {
  interface none;
  composition
    c1 : Comp1;
    c2 : Comp2;
  attachments
    c1.r1(a,b), c1.r2(c) <> c2.r3(a,b,c);
  }
```

```
component Comp1 {
  interface
    r1 : Rol1(x,y);
    r2 : Rol2(z);
  }
component Comp2 {
  interface
    r3 : Rol3(u,v,w);
  }
```

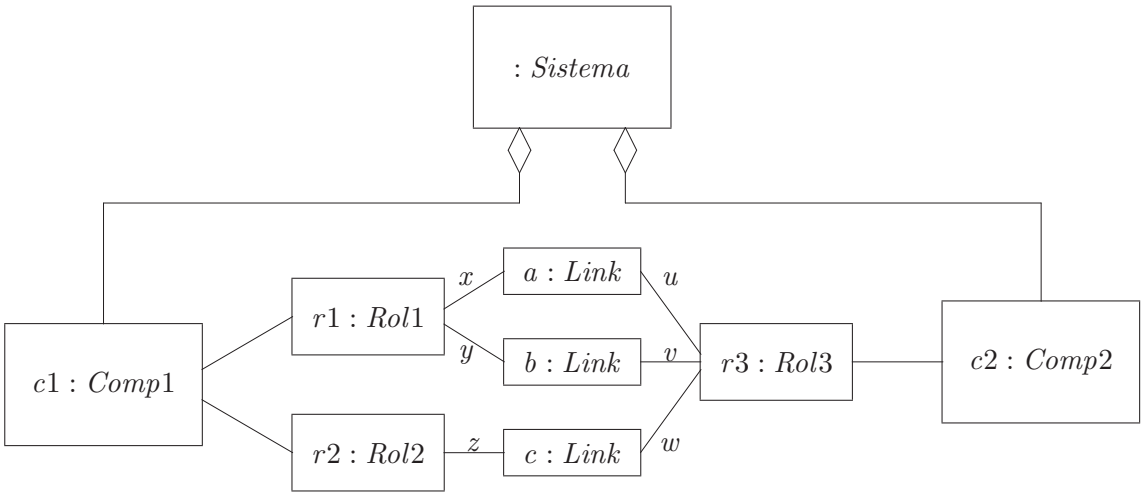


Figura 4.3: Ejemplo de sistema generado en Java.

En el sistema Java resultante de la generación existirán dos componentes, `c1` *y* `c2`*, que ofrecen métodos correspondientes a las computaciones a realizar cada vez que se envía o recibe un mensaje a través de uno de los enlaces utilizados en sus roles. Denominaremos a estos métodos de la misma forma que a las acciones de entrada y salida correspondientes, por ejemplo,* `xEXC()` *(del inglés* exclamation mark*) para referirnos a la acción* $x!()$*, o* `xQUE()` *(del inglés* question mark*) para referirnos a* $x?()$*. Estos métodos del componente tienen como objetivo, respectivamente, la elaboración de los datos que van a ser enviados en una acción de salida, o el procesamiento de los datos recibidos en una acción de entrada.*

Existirán así mismo tres procesos o flujos de control, los correspondientes a los roles `r1`*,* `r2` *y* `r3`*. Cada una de estos flujos seguirá el patrón de coordinación indicado en las correspondientes clases de rol* `Rol1`*,* `Rol2` *y* `Rol3`*, encargándose de la emisión y recepción de mensajes a través de los enlaces y de invocar el método correspondiente de los componentes cada vez que se realiza una acción.*

Por último, los objetos enlace `a`*,* `b` *y* `c` *son los que permiten comunicar unos roles con otros y ofrecen operaciones de lectura y escritura que son invocadas por los roles.*

El esquema de generación de código presentado lleva a una implementación natural del prototipo a partir de su especificación en LEDA. Este prototipo será de carácter concurrente y posiblemente distribuido y en él cada componente estará asociado a varios flujos de control, uno por cada uno de sus roles.

De acuerdo con esto existirán tantos flujos de control como roles haya en el sistema. Los roles de componentes distintos se coordinan por medio de la ejecución de acciones de lectura y escritura de los enlaces compartidos entre ellos, mientras que los roles de un mismo componente pueden coordinarse entre sí a través de la compartición de datos del componente.

Las cuatro clases citadas `Component`, `Role`, `Link` y `Action` son las clases fundamentales en torno a las cuales se construye el prototipo a partir de una especificación en LEDA. A continuación comentaremos brevemente las características principales de cada una de estas clases. Su implementación completa en Java se encuentra en el Apéndice C.

```
abstract class Component {
    String name;
    int pid;
  public Component(String name, int pid) {...}
  abstract public Object clone();
  abstract public void attach(Link links[]);
  public Object get(String field) {...}
  public void set(String field, Object value) {...}
  public Object perform(String method, Object prmtr) {...}
  public boolean test(Tau action) {...}
  }
```

Figura 4.4: Implementación en Java de la clase `Component`.

4.2.1 La clase `Component`

La Figura 4.4 muestra un esquema de la clase `Component`. Esta clase reúne las características básicas de los componentes, y de ella hereda cualquier clase de componentes que aparezca en una especificación. La característica más significativa de esta clase es que su implementación hace uso de las posibilidades de programación reflexiva que ofrece Java para acceder, desde los roles, a los atributos y métodos que implementa un componente.

Ya hemos indicado cómo los roles invocan métodos del componente cada vez que realizan operaciones de entrada o salida a través de los enlaces. Además de esto, un componente puede declarar variables que sirvan para coordinar entre sí sus diferentes roles. Por tanto, es necesario que las instancias de las clases de roles tengan acceso a las instancias de las clases de componentes a las que se asocian. Sin embargo, dado que en LEDA los roles pueden especificarse de forma separada a los componentes, por lo general, un rol no conoce cuál es la clase de componentes (heredera de `Component`) a la que va a estar finalmente asociada. Es posible, no obstante, declarar en el rol un atributo de clase `Component` al que será asignado posteriormente un objeto de la clase de componentes adecuada, mediante polimorfismo de datos. Sin embargo, esta solución no permite al rol acceder a los métodos o atributos de un componente (que era nuestro objetivo) puesto que la clase base `Component` no dispone de dichos métodos ni atributos.

Por este motivo, `Component` implementa, mediante técnicas reflexivas, métodos para acceder y modificar de forma genérica sus atributos (los métodos `get` y `set`, respectivamente), así como para invocar los métodos asociados a la realización de acciones de entrada o salida por los enlaces (el método `perform`) y también para evaluar la posibilidad de ejecutar una acción local (el método `test`).

Una ventaja añadida de esta solución, para el caso de los métodos asociados a acciones de entrada o salida, es que no es necesario generar métodos vacíos para representar a estas acciones, sino que en caso de no estar definido se muestra un mensaje genérico que informa de la invocación del método. Al ir refinando el prototipo se implementarán de forma adecuada estos métodos, dotando al componente de la funcionalidad requerida.

4.2.2 La clase `Role`

La clase `Role`, cuya implementación se muestra de forma resumida en la Figura 4.5, reúne las características básicas de los roles, y de ella hereda cualquier clase de roles que aparezca en el

```
class Role implements Runnable {
    String name;
    int pid;
    Thread t;
    Component component;
    State state;
    ActionList offers;
    Action commitment;
  public Role(String name, int pid, Component component) {...}
  public void attach(Link links[]) {...}
  public void run() {...}
  public synchronized Action getCommitment() {...}
  public synchronized boolean setCommitment(Action a) {...}
  public synchronized void resetCommitment() {...}
  int select() {...}
  void tryCommitment() {...}
  Object perform() {...}
  void removeOffers() {...}
  }
```

Figura 4.5: Implementación en Java de la clase `Role`.

sistema. Entre sus atributos destaca el llamado `component`, que hace referencia al componente poseedor del rol. Este atributo se declara de clase `Component`, tal como indicamos al hablar de esta última clase.

Dado que los roles son los encargados de los aspectos de coordinación del sistema generado, por cada instancia de la clase `Role`, es decir, por cada rol del sistema, tendremos un flujo de control o hebra, por lo que `Role` implementa el interfaz Java `Runnable` y declara un método `run` que es el que se ejecuta al activar la hebra asociada al rol. Este método `run` habrá de implementar el comportamiento descrito por el rol.

La especificación en LEDA de un rol se realiza, como hemos visto, por medio de la definición de una serie de agentes en el cálculo π, donde cada uno de dichos agentes se corresponde con un *estado* del rol. El patrón de comportamiento descrito por el rol indica como éste va cambiando de estado en función de su interacción con el resto de roles del sistema. En cada estado, el rol puede reaccionar de forma distinta ante el mismo evento. Por este motivo, la implementación que hemos elegido para los roles sigue el patrón de diseño *Estado* descrito en [Gamma et al., 1995]. De este modo, y a través del atributo `state`, cada rol va a tener asociado un estado que será subclase de la clase abstracta `State`, tal como muestra la Figura 4.6, y existirá una subclase estado por cada uno de los agentes que incluya la especificación del rol.

En cada uno de estos estados, subclases de `State`, se implementa el método `spec`, que es el que contendrá finalmente la especificación del estado como una serie de acciones que se realizan sobre los enlaces del rol. El resultado del método `spec` será el nuevo estado al que pasa el rol después de ejecutar las acciones indicadas en el actual. Por cada rol existirá un estado inicial en el cual comenzará la ejecución del rol cuando se activa el método `run`. Existe además una subclase especial de `State`, la clase `Zero`, que se identifica con la inacción del cálculo π y es utilizada para marcar la terminación de un rol.

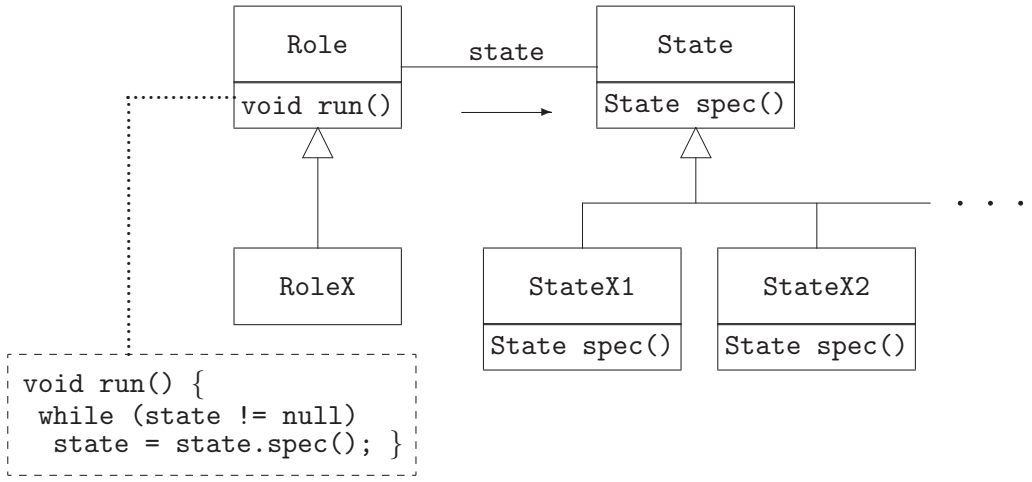


Figura 4.6: Implementación de los roles por medio del patrón *Estado*.

4.2.3 La clase `Link`

La clase `Link`, cuya implementación se muestra de forma resumida en la Figura 4.7, modela y encapsula las características de los enlaces del cálculo π, de forma que el mecanismo de comunicación utilizado no influye en la implementación de los componentes ni en la de los roles. Estos enlaces permiten establecer comunicaciones uno a uno, uno a muchos, o muchos a muchos, de tal forma que los componentes involucrados en un diálogo no tengan que ser conscientes de ello.

Entre los atributos de la clase `Link` destaca `data`, utilizado para almacenar temporalmente el objeto escrito en el enlace por una acción de salida, hasta que sea recuperado por el rol que realice la lectura correspondiente. Otro atributo interesante es `number`, que indica el número de roles que están haciendo uso del enlace, lo que como veremos será utilizado para realizar

```
class Link {
    String name;
    Object datum;
    boolean written;
    ActionList offers;
    int nroles;
    LinkImp imp;
  public Link(String name) {...}
  public Link(String name, Object datum) {...}
  public void incr() {...}
  public void decr() {...}
  public synchronized boolean write(Object datum) {...}
  public synchronized Object read() {...}
  }
```

Figura 4.7: Implementación en Java de la clase `Link`.

```
abstract class Action {
    Role role;
    int number;
    int sign;
  public Action(Role role, int number, int sign) {...}
  abstract public String toString();
  abstract public Object perform();
  }
```

Figura 4.8: Implementación en Java de la clase `Action`.

acciones de difusión a través del enlace. El atributo `written` indica si un enlace ha sido escrito (y todavía no leído) o no.

Además de estos atributos, la clase `Link` ofrece métodos para la escritura (`write`) y lectura (`read`) del enlace. El método `write` devuelve el valor lógico verdadero si el enlace ha logrado ser escrito (para lo cual no debe contener datos previamente) mientras que `read` devuelve en dato leído en caso de que haya podido realizarse la operación de lectura sobre el enlace (para lo cual éste debe contener datos previamente).

Durante la ejecución del prototipo es posible la creación dinámica de nuevos enlaces, al igual que la transmisión de enlaces entre procesos, de forma que se modifique la topología de comunicación del sistema. Así, la acción de salida ligada *(x)a!(x)*, consistente en la creación de un nuevo enlace x que es enviado a través de un enlace a ya existente, se implementa en Java de la forma siguiente:

```
x = new Link("x");
a.write(x);
```

mientras que el receptor del mensaje —que realiza la acción de entrada complementaria *a?(y)*— debe ejecutar la expresión:

```
y = a.read();
```

logrando acceso al enlace x, que puede ser utilizado en futuras comunicaciones entre ambos procesos.

4.2.4 La clase `Action`

Finalmente, la clase `Action`, cuya implementación se muestra de forma resumida en la Figura 4.8, modela las acciones atómicas que pueden ser realizadas por los roles. Entre sus atributos destaca `sign`, que indica el signo de la acción. Esta clase declara además el método abstracto `perform`, que es el encargado de ejecutar la acción. Este método es implementado en las subclases de `Action` dependiendo del tipo de acción de que se trate. Existen cinco tipos de acciones en LEDA, que se implementan como subclases de `Action`:

- **Acciones internas.** Estas acciones están modeladas por la clase `Tau` y se caracterizan por no tener signo ni estar asociadas a ningún enlace. Su ejecución consiste simplemente en la invocación del método `tau` asociado a dicha acción en el componente correspondiente.

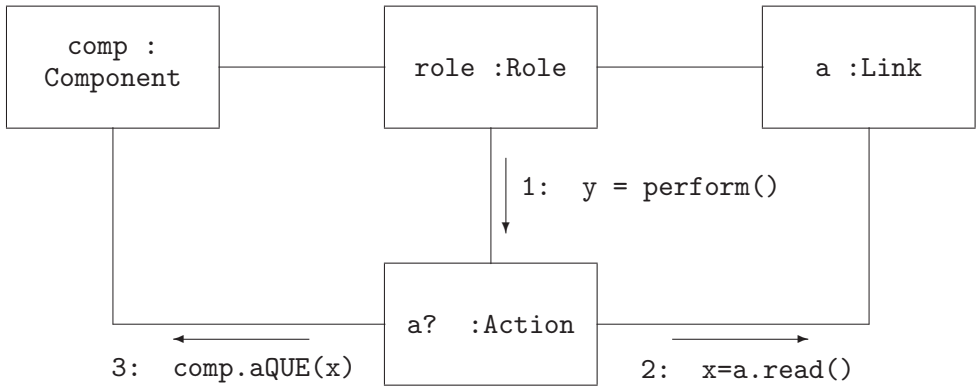


Figura 4.9: Ejecución de una operación de lectura $a?(y)$ por parte de un rol.

- **Acciones de salida.** Las acciones de salida están modeladas por la clase `Output`, y se caracterizan por tener signo positivo (+1). Al crear una de estas acciones se debe indicar cuál es el enlace *sujeto* de la acción (aquél por el cual se va a realizar la salida), así como cuál es el *objeto* de la misma (es decir, el dato que se va a enviar a través del enlace sujeto). La ejecución de una acción de salida $a!(x)$ implica en primer lugar la invocación del método `aEXC()` del componente correspondiente, que devuelve un valor que se asigna al dato `x` que va a ser enviado a través del enlace `a`, seguida a continuación de la escritura del enlace por medio del método `write` de la clase `Link`.

- **Acciones de entrada.** Las acciones de entrada están modeladas por la clase `Input` y se caracterizan por tener signo negativo (-1). Al crear una de estas acciones se debe indicar cuál es el enlace sujeto de la misma.

  La Figura 4.9 muestra el diagrama de colaboración que corresponde a la ejecución de una acción de entrada $a?(y)$ por parte de un rol. Para ello, el rol encarga a la acción que se ejecute, invocando el método `perform`. Esta ejecución consiste en la realización de una operación de lectura sobre el enlace `a`, invocando el método `read()` y recibiendo como resultado un enlace `x` leído en `a`. A continuación, la acción invoca sobre el componente asociado al rol el método `aQUE(x)` por el que le transmite el dato leído para que sea procesado por el componente. Por último, el dato leído es devuelto al rol, que lo asigna al enlace `y`.

- **Acciones de salida múltiple (difusión).** Las acciones de difusión o salida múltiple están modeladas por la clase `MultiOutput`, y se caracterizan por tener signo positivo, mayor que uno, con un valor determinado por el número de roles que deben leer del enlace. Al crear una de estas acciones se debe indicar cuál es el enlace *sujeto* de la acción (aquél por el cual se va a realizar la salida), así como cuál es el *objeto* de la misma, es decir, el dato que se va a enviar a través del enlace sujeto.

  La ejecución de una acción de difusión $a[*]!(x)$ a `n` receptores implica en primer lugar la invocación del método `aBRD()` (del inglés *broadcast*) del componente correspondiente y la asignación de su resultado a `x`, seguida de una serie de operaciones elementales de escritura de enlaces, tal como se recoge en la interpretación semántica de estas operaciones en el cálculo π, detallada en la Sección 3.9.1.

La determinación del número de receptores de la difusión se realiza a partir del atributo `nroles` del enlace, que es incrementado cada vez que se utiliza el enlace en una conexión entre roles. A continuación se crea un enlace $\texttt{l}_i$ para cada uno de los `n` receptores, así como un enlace de acuse de recibo `ack` común para todos ellos. Los `n` enlaces $\texttt{l}_i$ son enviados a los receptores a través del enlace `a` (que era el sujeto de la difusión), con lo que cada receptor obtendrá uno de estos enlaces. Acto seguido, el dato `x`, objeto de la difusión, es transmitido a los receptores a través de cada uno de los enlaces $\texttt{l}_i$, y se difunde también de esta forma el enlace `ack`. Por último, se recogen `n` mensajes de acuse de recibo a través del enlace `ack`, proveniente cada uno de uno de los receptores. Esta implementación de la acción de difusión asegura que ni el rol escritor ni los lectores finalizan la ejecución de la acción antes de que el enlace x, objeto de la difusión, haya sido transmitido a todos los lectores.

- **Acciones de entrada múltiple.** Las acciones de entrada múltiple son el contrapunto de las acciones de difusión y están modeladas por la clase `MultiInput`, teniendo signo negativo (-1). Al crear una de estas acciones se debe indicar cuál es el enlace sujeto de la misma.

  Tal como se indica en la Sección 3.9.1, la ejecución de una acción de entrada múltiple $a[]?(y)$ implica la realización de una serie de operaciones elementales de lectura de enlaces, consistentes en la recepción de un enlace exclusivo `l` a través del enlace `a` (que está compartido con el resto de los receptores), seguida de la recepción del objeto de la difusión `x` y de un enlace de acuse de recibo `ack` a través del enlace `l`. Por último, el receptor envía un acuse de recibo a través de `ack`.

  Una vez realizada la lectura, y suponiendo que el valor obtenido en la misma es x, se realiza la invocación del método `aQUE(x)` del componente correspondiente. Finalmente, se devuelve el valor `x` leído.

Por último, señalar que las acciones de entrada o salida *poliádica*, es decir, aquéllas que utilizan varios enlaces como objetos de la acción, no tienen un tratamiento especial por parte de la clase `Action`. El generador de código descompone estas acciones poliádicas en acciones moleculares de acuerdo con las definiciones:

$$a!(x_1,\ldots,x_n) \stackrel{\text{def}}{=} (w)\ a!(w)\ w!(x_1)\ \ldots\ w!(x_n)$$
$$a?(y_1,\ldots,y_n) \stackrel{\text{def}}{=} a?(w)\ w?(y_1)\ \ldots\ w?(y_n)$$

4.3 El mecanismo de compromiso

El cálculo π es un álgebra de procesos síncrona, en el sentido de que para que un proceso realice una determinada acción, de entrada o de salida, es preciso que exista otro proceso en el sistema, compuesto en paralelo con el primero, que realice la acción complementaria. Esto es lo que indica la regla fundamental del sistema de transiciones del cálculo que recordamos aquí:

$$a!(x).P \ \mid\ a?(y).Q \ \longrightarrow\ P \ \mid\ Q\{x/y\}$$

En un momento determinado, un proceso puede presentar varias acciones posibles, por ejemplo combinadas mediante el operador de composición alternativa. La elección entre una u otra de estas alternativas es lo que se denomina compromiso (*commitment*) y dependerá de la

disponibilidad de acciones complementarias en algún otro proceso del sistema, tal como muestra el ejemplo:

$$(\ a!(x).P1 + b!(y).P2 + c!(z).P3\ )\ \mid\ b?(w).Q \longrightarrow P2\ \mid\ Q\{y/w\}$$

donde vemos que la alternativa elegida por el primer proceso será siempre la segunda, debido a la presencia del segundo proceso. En caso de haber varios emparejamientos posibles, se comprometerá uno cualquiera de ellos de forma no determinista.

La implementación de este mecanismo de compromiso no es complicada, siempre y cuando diseñemos un control centralizado en la aplicación. Este sería, por ejemplo, el caso de un simulador secuencial, que disponga de la especificación de todos los procesos existentes en el sistema, como las herramientas ya citadas MWB [Victor, 1994] o e-pi [Henderson, 1998]. Sin embargo, la situación se complica si pretendemos realizar una implementación concurrente o distribuida del simulador, como es nuestro caso, en el que cada rol se ejecuta en un flujo de control distinto, de forma que la combinación de estos flujos de control modela la composición paralela de los roles. En esta situación, para comprometer una pareja de acciones, debemos poner de acuerdo en dicho compromiso a dos procesos que se ejecutan en flujos de control distintos, de forma que ninguno de los roles se comprometa a realizar una acción si otro no se compromete a la complementaria. Además, la ejecución de dichas acciones debe realizarse de forma síncrona en ambos procesos. Por otro lado, para elegir de forma no determinista entre varios compromisos en el sistema debemos analizar cuáles son los emparejamientos posibles de acciones complementarias entre los roles que se ejecutan en el conjunto de flujos de control del sistema.

Para resolver esta cuestión podemos optar por utilizar una versión asíncrona del cálculo. Tal como hemos mencionado en la Sección 1.5.1 existen diversas versiones asíncronas de este álgebra de procesos, en las que, básicamente, la asincronía se logra al prescindir del carácter prefijo de la acción de salida, que pasa a realizarse de forma concurrente al proceso en el que se encuentra. Así, el agente:

$$a!(x).P\ \mid\ a?(y).Q$$

se interpreta en el cálculo asíncrono como:

$$(a!(x).\mathbf{0}\ \mid\ P\ )\ \mid\ a?(y).Q$$

de forma que el proceso de la izquierda puede proceder a P aún cuando no se haya realizado el emparejamiento de las acciones $a!(x)$ y $a?(y)$. Obsérvese el tratamiento asimétrico que damos en este caso a las acciones de entrada y salida. En efecto, las acciones de entrada siguen siendo prefijas puesto que, en el ejemplo, no tiene sentido que el segundo proceso proceda sin antes realizar la acción de entrada $a?(y)$ con la consiguiente sustitución en Q del nombre y por el nombre recibido.

La implementación concurrente de un simulador del cálculo asíncrono no presenta especiales dificultades, al no existir necesidad de establecer mecanismo alguno de compromiso: el proceso que deba realizar una acción de salida escribirá en el enlace correspondiente y procederá con independencia de su contexto, mientras que el que deba realizar una acción de entrada bloqueará hasta que dicho enlace haya sido escrito.

Aún en la versión asíncrona del cálculo es posible representar las comunicaciones síncronas como acciones derivadas [Boudol, 1992, Boudol, 1994]. Así, las acciones prefijas del cálculo síncrono pueden definirse como acciones derivadas en el asíncrono de la siguiente forma:

$$a!(x).P \stackrel{\text{def}}{=} (p)(\ a!(p).\mathbf{0} \mid p?(q).(\ q!(x) \mid p?(q).P\ )\ )$$
$$a?(y).Q \stackrel{\text{def}}{=} a?(p).(q)(\ p!(q).\mathbf{0} \mid q?(y).(\ p!(q).\mathbf{0} \mid Q\ )\ )$$

(donde p y q no aparecen como nombres libres en P ni Q).

Sin embargo, esta equivalencia entre el cálculo síncrono y el asíncrono se limita sólo al llamado *mini*-cálculo π [Milner, 1992], un subconjunto del cálculo π que presenta propiedades de interés, pero en el que se prescinde, entre otros, del operador de composición alternativa (+). Esta limitación es la causa de que los lenguajes derivados del cálculo asíncrono, como Pict [Pierce y Turner, 1999] o Piccola [Lumpe, 1999], carezcan del operador de composición alternativa.

Trabajos posteriores a los de Boudol proponen diversas formas de codificación del operador de composición alternativa en el cálculo asíncrono [Nestmann y Pierce, 1996], resolviendo la codificación de alternativas guardadas únicamente por acciones de entrada o de salida (no mezcladas), e incluso de alternativas *mixtas*, pero en este último caso no se asegura ya sea la justicia, ya la convergencia del algoritmo de compromiso. Finalmente, se ha demostrado que la expresividad del cálculo asíncrono es inferior a la del síncrono [Palamidessi, 1997], en particular en sistemas formados por procesos simétricos, en los que no es posible resolver el problema de *elección de líder*, lo que se traduce en que no se garantiza que los procesos puedan alcanzar un compromiso sobre la acción a realizar en un lapso finito de tiempo.

Por tanto, la implementación de un mecanismo distribuido de compromiso, necesario para la generación de código a partir de especificaciones en LEDA ha de tener en cuenta estas limitaciones, dado que nuestro lenguaje incorpora el operador de composición alternativa y además de forma no restringida, es decir, permitiendo la combinación de alternativas mixtas, con guardas formadas por acciones de entrada, salida e internas.

El mecanismo de compromiso implementado reside fundamentalmente en la clase `Role` y consta de tres fases:

- **Oferta.** Cada rol publica una *lista de ofertas* con las acciones a las que puede comprometerse en un momento determinado. Estas acciones se corresponden con las que figuran en las guardas de la alternativa no determinista que esté por realizar el rol. Las ofertas se anotan no sólo en el rol, sino también en los respectivos enlaces (es decir, cada enlace contiene referencias a los roles dispuestos leer o escribir por él), utilizando sendos atributos `offers` existentes en las clases `Role` y `Link`.

- **Selección.** A continuación, el rol inicia un proceso de selección (implementado por el método `select`) en el que trata de comprometer alguna de las acciones ofrecidas. Para ello, selecciona de manera aleatoria una de las acciones de su lista de ofertas, y comprueba, a través del enlace correspondiente, si algún otro rol ofrece la acción complementaria. Si es así, se intenta comprometer a ambos roles en la ejecución del par de acciones y finaliza el proceso de selección. Si no está disponible la acción complementaria, o si no es posible el compromiso (por estar ya el otro rol comprometido en otra acción) el intento fracasa, eligiéndose de nuevo una acción de la lista. Un rol continuará en esta fase de selección hasta que logre comprometerse (bien por sí mismo o bien indirectamente porque otro rol comprometa a éste). La acción comprometida se almacena en el atributo `commitment`.

- **Ejecución.** Una vez comprometida una acción, se elimina la lista de ofertas del rol y procede a la ejecución de la misma, por medio del método `perform`.

Para lograr el correcto funcionamiento de este algoritmo, que es ejecutado de forma concurrente por todos los roles del sistema, debemos asegurar el acceso en exclusión mutua a la lista de ofertas (`offers`) de roles y enlaces, así como al atributo `commitment` que indica el compromiso o no de un rol. Esto se logra haciendo uso de métodos sincronizados (*synchronized*) para consultar y modificar estos atributos.

Por otra parte, el establecimiento del compromiso debe realizarse forma atómica en los dos roles afectados. Para ello este compromiso se lleva a cabo por medio de un método sincronizado `commit` en un objeto especial `Lock` de forma que, o bien ambos roles, o bien ninguno de los dos, resulten comprometidos.

Por último, las acciones internas siguen un mecanismo de compromiso especial, debido a que estas acciones no implican el uso de ningún enlace e involucran a un solo rol. Si una acción interna es elegida en la fase de selección, el rol podrá comprometerse a realizar dicha acción siempre y cuando no haya sido ya comprometido por otro. No obstante, dado el carácter interno de estas acciones, la decisión final de comprometerse o no depende del componente. Para modelar esta decisión se invoca el método `test` de la clase `Component`, que devuelve un valor lógico indicando si el componente está dispuesto a realizar o no la acción. Inicialmente este método devuelve el valor lógico verdadero para cualquier acción interna, pero esto puede ser modificado durante el desarrollo del sistema a partir del prototipo variando influyendo sobre la probabilidad de elección entre varias acciones internas alternativas, tal como veremos en diversos ejemplos en la Sección 4.5.

El siguiente ejemplo ilustra el funcionamiento del mecanismo de compromiso:

Ejemplo 4.2 *Supongamos un rol $R(a, b)$, especificado en LEDA tal como muestra la Figura 4.10 (izquierda). En ella vemos cómo el rol puede realizar tres acciones —$a!(x)$, $b?(y)$ y t— que aparecen como guardas en una alternativa no determinista.*

```
role R(a,b) {                    public State spec() {
                                   new Output(1,a,x);
 ...                               new Input(2,b);
                                   new Tau(3);
 spec is                           int choice = select();
                                   switch(choice) {
  a!(x). R1(a,b)                     case 1 :  perform();
                                               return new R1(a,b);
 +b?(y). R2(a,b)                     case 2 :  Link y = perform();
                                               return new R2(a,b);
 +t. R3(b,a);                        case 3 :  perform();
                                               return new R3(b,a);
 ...                               }
 }                               }
```

Figura 4.10: Traducción a Java de la especificación de un rol.

La parte derecha de la figura corresponde a la traducción resultante en Java de la especificación del comportamiento del rol, y muestra concretamente la implementación del método

`spec` *de la clase* `R`*, subclase de* `State`*, que tal como indicamos en la Sección 4.2.2, es donde se recoge la especificación de dicho comportamiento.*

En primer lugar, se crean las tres acciones correspondientes a las guardas de la alternativa que aparece en el rol, indicando para cada una de ellas el tipo de la acción (como subclase de `Action`*), los enlaces que utiliza, y un número correlativo que nos servirá para identificar las distintas ramas de la alternativa. Al crearse, estas acciones pasan automáticamente a formar parte de la lista de ofertas del rol, así como de los enlaces correspondientes. Esta parte del código se corresponde, por tanto, con la fase de oferta del mecanismo de compromiso.*

A continuación entramos en la fase de selección, representada por el método `select`*. Tal como hemos indicado, en este método el rol seleccionará de manera aleatoria una de las tres acciones y tratará de comprometerla. Si no fuese posible, elegirá otra, y así hasta que logre un compromiso. En el caso de las dos primeras acciones —de entrada y salida, respectivamente— necesitará para ello de la concurrencia de otro rol dispuesto a realizar la acción complementaria. En el caso de la tercera, la acción interna, bastará con su propio compromiso, siempre y cuando el método de prueba asociado a la acción tenga resultado positivo. En cualquier caso el método* `select` *finaliza comprometiendo una acción (que se almacena en el atributo* `commitment` *del rol) y devolviendo el número correlativo asignado a la misma.*

Este número es utilizado en la fase de ejecución, representada en el código por la instrucción `switch` *que, dependiendo de la alternativa seleccionada, realizará unas operaciones u otras. En caso de que la acción comprometida sea la primera —la acción de salida* $a!(x)$*—, el método* `perform` *ejecutará dicha acción y a continuación el rol evolucionará al estado* $R1(a, b)$*, para lo que el método* `spec` *finaliza devolviendo una nueva instancia de la clase* `R1`*, cuyo método* `spec` *pasará a ejecutarse a continuación.*

Si, por el contrario, la acción comprometida es $b?(y)$*, estaremos entonces en la segunda de las alternativas del* `switch`*. El método* `perform` *ejecutará dicha acción y su resultado se asignará al enlace* `y`*. A continuación el rol evolucionará al estado* $R2$*, devolviendo el método* `spec` *una instancia de dicha clase.*

Por último, si el mecanismo de compromiso elige la tercera de las alternativas, se ejecuta la acción interna con el método `perform`*, al igual que en los casos anteriores y el rol evolucionará al estado* $R3$*, realizando la permutación de los enlaces* a *y* b *de acuerdo a lo indicado en la especificación en LEDA.*

Obsérvese que la continua creación y destrucción de instancias de `State` *en cada cambio de estado puede evitarse fácilmente utilizando el patrón de diseño Singular, descrito en [Gamma et al., 1995], que nos permitiría utilizar una única instancia permanente para cada una de las clases estado.*

Como conclusión de esta sección podemos señalar que el mecanismo de compromiso que acabamos de describir es *justo*, en el sentido de que todas las acciones que figuran como guardas en una alternativa no determinista tienen la misma posibilidad de resultar elegidas y ser comprometidas. Además su implementación se ha realizado de forma que se logre la exclusión mutua sobre los datos compartidos entre los distintos roles (la lista de acciones y los atributos `commitment`) y se evite además la posibilidad de interbloqueos, para lo que todos los métodos definidos como sincronizados acaban en un tiempo finito.

Sin embargo, y tal como se establece en [Palamidessi, 1997], existe la posibilidad de que el algoritmo de compromiso diverja, es decir, que no acabe en un tiempo finito. La divergencia puede producirse en dos situaciones.

En primer lugar, el mecanismo de compromiso diverge si el conjunto de roles bloquea. En efecto, no se ha contemplado un mecanismo de detección distribuida de bloqueo, por lo que puede suceder que los roles permanezcan de forma indefinida tratando de lograr un compromiso que no exista. Sin embargo, debe tenerse en cuenta que el análisis de la especificación haciendo uso de la relación de compatibilidad descrita en el Capítulo 2 asegura la ausencia de bloqueos en las conexiones de los roles del sistema, por lo que esta situación no se producirá de haberse realizado previamente dicho análisis de compatibilidad. En cualquier caso, existen numerosos algoritmos de detección distribuida de interbloqueo [Knapp, 1987, Singhal, 1989], como por ejemplo el propuesto por Chandy y colaboradores [Chandy et al., 1983], que pueden ser incorporados como parte del mecanismo de compromiso.

En segundo lugar, y tal como se apunta en [Palamidessi, 1997] al tratar el problema de la elección de líder en las versiones asíncronas del cálculo π, el mecanismo de compromiso implementado puede divergir en sistemas formados por roles idénticos. Esto sucedería en el caso de que, de forma simultánea en todos ellos, cada rol intentase comprometer a otro en una acción de entre las que oferta. Estos intentos de compromiso fracasarían debido a que cada rol rechazaría el compromiso que le es ofrecido, al intentar alcanzar el que él mismo propone, con lo que finalmente todos los roles darían marcha atrás en el intento de compromiso, repitiéndose este proceso una y otra vez. Sin embargo, para que esta situación se produjese en la práctica, además de la existencia de roles idénticos, debería producirse una ejecución perfectamente sincronizada del mecanismo de compromiso por parte de cada uno de ellos.

4.4 Generación de código

Una vez descritas las clases que implementan la funcionalidad común a todos los prototipos de los sistemas especificados en LEDA, mostraremos a continuación cómo se realiza la generación de código a partir de los componentes y roles especificados por el diseñador. Para ello, utilizaremos como ejemplo el sistema de subastas descrito en el Sección 3.10. La implementación completa generada a partir de este sistema se puede encontrar en el Apéndice C.

4.4.1 Componentes

El esquema de traducción utilizado transforma cada clase de componentes de una arquitectura en LEDA en una clase Java que hereda de `Component`. Por tanto, cada instancia de componente se traducirá por una instancia de la clase Java correspondiente.

Como hemos visto, estas clases son *pasivas*, en el sentido de que no invocan métodos de otras clases del sistema, sino que sus métodos van a ser invocados por los roles. La interfaz de las clases componente constará de un método por cada enlace utilizado por el componente como sujeto de acciones de entrada y otro por cada enlace de utilizado en acciones de salida.

La implementación de estos métodos se corresponde con el procesamiento que debe realizarse tras recibir un mensaje (métodos sobre acciones de entrada por un enlace), o previamente al envío de mensajes (métodos sobre acciones de salida). En nuestro enfoque los componentes no incluyen ningún tipo de información acerca de cómo sus computaciones internas afectan al resto del sistema. Por tanto, un componente nunca toma la iniciativa de enviar un mensaje, sino que será controlado por uno o varios objetos rol que implementarán el comportamiento asociado al componente. En las primeras fases de desarrollo del sistema no es necesario implementar dichos métodos, pudiendo simularse el comportamiento global del sistema sin necesidad de introducir errores propios de la manipulación de datos.

Al crearse un componente, se inicializan sus variables locales y se crean además los roles declarados en su interfaz, junto con los subcomponentes que figuran en la sección de composición. Posteriormente se ejecutará el método `attach`, que tiene como finalidad el conectar el componente con el resto del sistema y en el que se reciben como parámetros una serie de enlaces que son utilizados para la conexión de los roles declarados en la interfaz del componente, lo que se lleva a cabo mediante la invocación de los métodos `attach` de dichos roles. En este método `attach` del componente se realiza además la interconexión de los subcomponentes, si los hubiese, como veremos en la Sección 4.4.4.

Ejemplo 4.3 *Consideremos como ejemplo el componente* Subastador *del sistema de subastas descrito en la Sección 3.10. A partir de la especificación en LEDA de esta clase de componentes, se genera en Java una clase* `Subastador` *con la siguiente implementación:*

```
class Subastador extends Component {
  public int max;
  Role conectar;
  Role subastar;
  public Subastador(String name,int pid) {...}
  public Object clone() {...}
  public void attach(Link links[]) {...}
  }
```

Como vemos, `Subastador` *se define como subclase de* `Component`*, heredando las características de ésta, en particular todo lo relativo al acceso mediante programación reflexiva a los métodos y atributos del componente. Además, la clase* `Subastador` *declara los atributos que aparecen en la sección de variables de la especificación del componente en LEDA (en este caso el atributo* `max`*, que indica el número de postores conectados en cada momento a la subasta), así como los roles que figuran en la interfaz del componente, que se declaran como pertenecientes a la clase base* `Role`*.*

El constructor de la clase inicializa la variable local `max` *y crea los roles* `conectar` *y* `subastar`*, cada una de ellos como instancia de la clase correspondiente. La conexión de estos roles con los del* `GeneradorDePostores` *se realiza en el método* `attach`*, mediante la compartición de los objetos enlace que figuran como argumentos de este último.*

4.4.2 Roles y adaptadores

En la especificación en LEDA, los roles indican el comportamiento de sus respectivos componentes, por tanto, serán los encargados de implementar este comportamiento. Los roles harán las funciones de conectores de unos componentes con otros, realizando las operaciones de lectura y escritura por los enlaces adecuados y en el orden correcto, y también se encargarán de invocar las funciones que implementan los componentes.

Respecto a los adaptadores, recordemos que éstos son elementos similares a los roles que son utilizados para adaptar la interfaz de dos o más componentes para lograr que sean compatibles. Mientras los roles figuran en la sección de interfaz de un compuesto, los adaptadores figuran en la sección de composición, pero dejando a un lado estas diferencias, la generación de código para estas clases adaptadores es exactamente igual que para los roles por lo que, en esta sección nos referiremos a ambos en conjunto.

Cada clase de roles o adaptadores que figure en la especificación en LEDA de un sistema da lugar a una clase Java heredera de `Role`. En esta clase se declaran las variables locales que figuren en la especificación del rol. Además, se genera una clase heredera de `State`, por cada uno de los estados que pueda presentar el rol, es decir, por cada uno de los agentes que se declaren en su especificación (incluyendo el primero de ellos, que se declara de forma no nombrada tras las palabras reservadas **spec is** y que define el estado inicial). Estas clases definen el comportamiento asociado a dicho estado al implementar el método abstracto `spec` declarado en la clase `State`.

Al crear una instancia de `Role`, ésta recibe del componente del que forma parte los enlaces que va a utilizar. De esta forma se consigue la interconexión de roles y adaptadores tal como se indica en la sección de conexiones del componente. Cada enlace dispone de un contador de uso que es incrementado aquí para asegurar el buen funcionamiento de las operaciones de difusión. En el constructor se asigna además el estado inicial de rol y se crea y activa la hebra donde se ejecutará el mismo.

Cada una de las clases que representan los diferentes estados dispondrá un constructor en el que también recibirá los enlaces a utilizar. Como ya hemos visto en el Ejemplo 4.2 esto permite realizar modificaciones en los enlaces en cada cambio de estado.

Ejemplo 4.4 *Consideremos como ejemplo el rol* Conectar *declarado inmerso dentro del componente* Subastador*. A partir de la especificación en LEDA de esta clase de roles, se generan en Java una serie de clases con la siguiente implementación:*

```
class SubastadorConectar extends Role {
  public SubastadorConectar(String name, int pid, Component component) {
    super("SubastadorConectar " + name, pid, component);
    state = new SubastadorConectarInicial(this);
    }
  }

class SubastadorConectarInicial extends State {
  Link conexion;
  public SubastadorConectarInicial(Role role) {
    super(role);
    }
  public void attach(Link links[]) {
    conexion = links[0];
    }
  public State spec() {
    perform(new Input(role,1,conexion));
    int max = ((Integer)role.component.get("max")).intValue();
    max++;
    role.component.set("max",new Integer(max));
    return this;
    }
  }
```

Lo más significativo aquí es la implementación del método `spec` *en la clase* `SubastadorConectarInicial` *que implementa el comportamiento del rol. Vemos en este método que se*

crea una acción de entrada correspondiente a la acción conexión?() *del rol, y que dicha acción se ejecuta (cuando se logra comprometer a otro rol en la acción complementaria) con el método* `perform`. *Posteriormente se accede, por medio de los métodos reflexivos* `get` *y* `set`, *a la variable* `max` *del componente* `Subastador`, *para incrementarla y se devuelve como estado siguiente el objeto actual, puesto que el rol* Subastador.Conectar *presenta un único estado.*

4.4.3 Creación dinámica de componentes

En LEDA es posible describir compuestos formados por un número indeterminado de componentes iguales, organizados en un *vector*[ ]. Inicialmente el vector estará vacío, y sus componentes se van creando en alguno de sus roles, mediante la instrucción **new** *vector*. Para implementar esto, el compuesto declara un atributo de clase `Vector` e implementa un método de creación que añade un componente al vector a partir de una semilla inicial.

Ejemplo 4.5 *Tomemos como ejemplo el componente* GeneradorDePostores, *que declara un vector de componentes* Postor. *A partir de su especificación en LEDA se genera la siguiente implementación:*

```
class GeneradorDePostores extends Component {
  Role conectar;
  public Vector postor;
  Component postorSeed;
  Link conexion;
  Link precio, puja, acuse, vendido, item, desc;
  public GeneradorDePostores(String name, int pid, Component postorSeed) {...}
  public GeneradorDePostores(String name, int pid) {...}
  public Object clone() {...}
  public void attach(Link links[]) {...}
  public void newPostor() {...}
  }
```

Como vemos, la clase `GeneradorDePostores` *declara un atributo* `postor`, *de clase* `Vector`, *que es inicializado en su constructor. Por otro lado, el método* `newPostor` *añade un nuevo elemento al vector, elemento que es creado mediante la clonación de una semilla* `postorSeed`. *El nuevo postor es conectado invocando su método* `attach`. *La llamada al método* `newPostor` *se realiza, de acuerdo a lo especificado en LEDA, en la implementación del rol* `conectar` *del componente* `generador`.

4.4.4 Interconexión de componentes

La especificación en LEDA de un compuesto describe la arquitectura del sistema o de un subsistema del mismo, e indica, en su sección de **attachments**, las relaciones existentes entre sus componentes integrantes mediante conexiones entre roles de dichos componentes. Por tanto, en los compuestos recae la responsabilidad de conectar los roles de sus componentes, dado que son ellos quienes tienen conocimiento acerca de dichas relaciones. Esta interconexión se realiza dentro del método `attach`, implementado por cada clase de componentes, y se lleva a cabo mediante la compartición de objetos enlace, tal como mostraba la Figura 4.3.

En el Capítulo 3 se describieron los distintos tipos de conexiones que existen en LEDA:

- **Estáticas.** Conectan de forma permanente dos o más roles de componentes distintos. En el método `attach` del compuesto en que figure la conexión se crean los objetos enlace utilizados en la misma, y estos enlaces se transmiten a los componentes relacionados por la conexión mediante la invocación del método `attach` de dichos componentes.

- **Reconfigurables.** En este caso aparecen varios roles alternativos en uno de los términos de la conexión, que ya no será permanente, sino que dependerá del valor que tome una determinada condición. En el método `attach` del compuesto donde figure la conexión reconfigurable se crearán tantos juegos de objetos enlace como posibilidades de conexión existan. Cada uno de los roles del término alternativo recibirá uno de estos juegos de enlaces, mientras que el rol que figure en el otro término recibirá todos los juegos y será el encargado de evaluar la condición y de establecer las conexiones adecuadas según el resultado de la misma. La evaluación de la condición (con la posible reconfiguración del sistema) se realiza cada vez que se produce un cambio de estado en el rol, al ejecutarse el método `attach` de la subclase de `State` correspondiente.

- **Múltiples.** La conexiones múltiples definen patrones de comunicación entre colecciones de componentes, y pueden implicar la compartición de nombres de enlaces (conectando de esta forma todos los componentes de la colección), o utilizar enlaces exclusivos para cada par de componentes conectados. En las conexiones múltiples compartidas existe la posibilidad de difundir un mensaje a todos los componentes conectados.

  Las conexiones múltiples con enlaces compartidos se realizan igual que las estáticas: el compuesto crea inicialmente los enlaces y los transmite a los componentes, a medida que éstos se van creando, tal como acabamos de ver en el Ejemplo 4.5. Por el contrario, en las conexiones que utilizan enlaces exclusivos se crea un juego de nombres por cada par de componentes conectado. Estos juegos de enlaces se almacenan en el compuesto y son utilizados para conectar los subcomponentes a medida que se van creando.

- **Exportación.** La exportación permite que un compuesto incorpore a su interfaz los roles de alguno de sus subcomponentes. Para realizar la conexión, el compuesto transmitirá los enlaces recibidos en el método `attach` a sus subcomponentes.

Ejemplo 4.6 *El componente* SistemaDeSubastas *está compuesto de un* Subastador *y un* GeneradorDePostores*. A partir de la especificación en LEDA de esta clase de componentes, se genera en Java una clase con la siguiente implementación:*

```
class SistemaDeSubastas extends Component {
  Component subastador;
  Component generador;
  Link conexion;
  Link precio, puja, acuse, vendido, item, desconexion;
  public SistemaDeSubastas(String name, int pid,
                           Component subastador,
                           Component generador) {...}
  public SistemaDeSubastas(String name, int pid) {...}
  public Object clone() {...}
  public void attach(Link[] links) {...}
  }
```

Como vemos, la clase `SistemaDeSubastas` *declara los componentes* `subastador` *y* `generador`*, así como los enlaces* `conexion`, `precio`*, etc., utilizados en las conexiones de su sección* **attachments***, y que conectan estáticamente los roles del* `subastador` *y del* `generador`*. En el constructor de la clase se crean estos enlaces y subcomponentes. Estos últimos se interconectan en el método* `attach` *por medio de la compartición de los citados enlaces.*

4.4.5 Instanciación

Una vez que hemos visto cómo se generan clases Java a partir de la especificación de componentes y conectores en LEDA, resta por describir cómo se realiza la instanciación de una arquitectura, lo que permite la obtención de un prototipo de la misma, mediante la creación de una instancia de la clase de componentes indicada. Así, la declaración:

instance *subasta : SistemaDeSubastas*;

a partir de la cual se genera la siguiente clase:

```
class InstanceSubasta {
  public static void main(String args[]) {
    new SistemaDeSubastas("sistema",0).attach(null);
    }
  }
```

crea una instancia de nuestro sistema de subastas que puede ser ejecutado en Java, sirviendo tanto de prototipo de la arquitectura especificada, como de punto de partida para el desarrollo del propio sistema de subastas, tal como veremos en la Sección 4.5.

4.4.6 Extensión y refinamiento

La extensión es el mecanismo del que dispone LEDA para adaptar roles y componentes a cambios en los requisitos, así como para fomentar su reutilización en sistemas distintos a aquéllos para los que fueron inicialmente especificados.

En el caso de los roles, la extensión permite realizar una redefinición completa del rol o de alguno de sus agentes, así como añadir nuevos agentes a un rol, o nuevas alternativas a alguno de los agentes del mismo. Las relaciones de herencia y extensión de roles definidas en el Capítulo 2 permiten comprobar que la extensión se ha realizado de forma que se preserve la compatibilidad del rol progenitor. Debido a las variadas posibilidades que ofrece la extensión de roles, durante la generación de código Java se realiza un *aplanamiento* de la herencia, de forma que la especificación del rol derivado se combina con la de su progenitor, dando lugar a una nueva clase de roles, en la que ya no hay herencia, y que es la que se traduce finalmente a Java. Por este motivo, la implementación en Java de la clase de roles derivada no es heredera de la clase progenitora, sino que es independiente de ésta.

Por su parte, la extensión de componentes permite la redefinición de los roles o subcomponentes de un compuesto, de forma que los nuevos tipos se deriven de los especificados en la clase progenitora. Así mismo, es posible añadir variables, roles, adaptadores o componentes a la clase progenitora. Estos mecanismos de redefinición y adición de nuevas características son similares a los utilizados en los lenguajes orientados a objetos, por lo que no resultaría complicada la

transcripción de la extensión en LEDA a la existente en Java. En efecto, para la adición de nuevas características al componente progenitor bastaría con declararlas en una clase Java que heredase de la que implementa al componente progenitor. Respecto a la redefinición, ésta ni siquiera afecta a la declaración de roles y componentes en el progenitor, ya que, como hemos visto en los ejemplos anteriores, éstos se declaran utilizando las clases base `Role` y `Component`. Sin embargo, sí que resultarían afectados el constructor de la clase (en el que habría que crear instancias de roles y componentes de los tipos redefinidos), y probablemente también el método `attach`, encargado de realizar las conexiones. No obstante esta facilidad de utilización de la extensión en Java para implementar la extensión en LEDA, por simetría con la solución propuesta para los roles, en la traducción a Java se realiza también el aplanamiento de la herencia de componentes.

Ejemplo 4.7 *En el sistema de subastas que hemos seguido como ejemplo a lo largo de todo en este capítulo se utiliza la extensión de roles y componentes para crear dos nuevas clases de componentes,* `SubastadorJusto` *y* `PostorLimitado` *que especializan al subastador y los postores especificados en un principio en dicho sistema. Las diferencias en el código generado para las clases progenitoras y las derivadas radican fundamentalmente en el método* `spec` *de los estados de los roles de dichos componentes, siendo las clases que implementan componentes y roles muy similares. La implementación completa de estas clases se encuentra en el Apéndice C.*

Con respecto al refinamiento, recordemos que éste se lleva a cabo mediante la instanciación de una arquitectura, reemplazando algunos de sus componentes por otros cuya clase sea derivada de la especificada en un principio. Para hacer posible este reemplazamiento, cada clase de componentes compuestos incluye, además de su constructor por defecto, otro en el que sus subcomponentes figuran como argumentos, como se puede observar en el código de los componentes del sistema de subastas que aparecen en los ejemplos anteriores. De esta forma, podemos obtener una instancia del compuesto en la que hayamos modificado la clase a la que pertenecen dichos subcomponentes.

Ejemplo 4.8 *Como ejemplo, consideremos un sistema de subastas en el que realicemos el refinamiento de la arquitectura implementada por la clase* `SistemaDeSubastas` *sustituyendo sus componentes por instancias de las clases* `SubastadorJusto` *y* `PostorLimitado`*. Dicha instanciación se declarara en LEDA de la siguiente forma:*

instance *subastaJustaLimitada : SistemaDeSubastas*[
subastador : SubastadorJusto;
generador.postor : PostorLimitado;
];

y da lugar a la generación de una clase:

```
class InstanceSubastaJustaLimitada {
  public static void main(String args[]) {
    new SistemaDeSubastas("sistema", 0,
                    new SubastadorJusto("subastador", 0),
                    new GeneradorDePostores("generador", 0,
                                        new PostorLimitado("postor", 0))
                    ).attach(null);
    }
  }
```

que podemos ejecutar para ver el efecto de las modificaciones realizadas en el sistema de subasta.

4.5 Desarrollo del sistema

Los esquemas de generación descritos en las secciones anteriores tienen como resultado un prototipo ejecutable del sistema, que se comporta tal como indica la especificación original realizada en LEDA y que preserva las propiedades de compatibilidad entre roles validadas sobre la especificación. Como hemos visto, la solución propuesta aplica el principio de separación de los aspectos de computación y coordinación en diferentes entidades (los componentes y los roles) con lo que se consiguen diversas ventajas, entre otras, un mayor grado de reutilización. En efecto, el enfoque seguido permite la reutilización y el refinamiento de forma separada de los aspectos de coordinación y computación del sistema, facilitando al diseñador la tarea de encontrar la mejor implementación para un determinado componente abstracto, modificando sólo la parte de procesamiento interno y reutilizando las partes relativas a la arquitectura y mecanismos de interacción del sistema.

4.5.1 Implementación de las computaciones

El objetivo es ahora realizar un desarrollo evolutivo e incremental del sistema, de forma que hagamos evolucionar el prototipo hasta convertirlo en el sistema final. Este desarrollo consistirá fundamentalmente en la implementación de los aspectos de computación de cada uno de los componentes del sistema, lo que se traduce en la codificación de los métodos de los mismos, de forma que los componentes desempeñen la funcionalidad adecuada. Recordemos que estos métodos están asociados a las acciones especificadas en los roles del componente y definen las computaciones que deben realizarse cada vez que se vayan a enviar o se hayan recibido datos a través de los enlaces que comunican unos roles con otros.

Para hacer que la transición del prototipo al sistema final sea suave, nos será de gran ayuda la capacidad de introspección de la que hemos dotado a los componentes, que permiten probar versiones incompletas del prototipo, en las que tengamos implementados algunos componentes y otros no. De esta forma combinamos las ventajas de tener un prototipo ya en las fases iniciales del desarrollo, nada más realizar la especificación del sistema, con un desarrollo gradual del mismo, en el que podemos realizar y probar la implementación de los componentes de forma separada y en el que el prototipo se va convirtiendo, poco a poco, en el sistema final.

Esta implementación podrá realizarse bien manipulando directamente el código generado, para añadir los métodos asociados a la lectura y escritura de enlaces, o bien refinando la especificación original, de forma que la completemos con aquellos aspectos computacionales que resulten relevantes para el sistema, para los cuales se podrá generar código de forma automática. Con esta última alternativa obtendríamos una especificación más completa del sistema, no sólo de su arquitectura, que ayudaría a la documentación del mismo, aunque como contrapartida, al incrementar el nivel de detalle de las especificaciones su análisis resultará computacionalmente más costoso.

Ejemplo 4.9 *En el Ejemplo 4.8 hemos utilizado la extensión y el refinamiento de los roles y componentes del sistema de subastas para incluir en su especificación aspectos computacionales que consideramos de especial interés (fundamentalmente porque afectan al comportamiento observable de los componentes recogido en sus roles, interviniendo en la toma de decisiones locales en los mismos). Así, por ejemplo, hemos añadido una variable* n *al rol* Subastar*, para llevar la cuenta del número de respuestas recibidas de los postores, y hemos especificado el manejo de dicha variable, que es utilizada para decidir la adjudicación o no de los artículos. La especificación refinada del sistema de subastas, en la que utilizamos esta nueva versión del subastador*

que denominamos SubastadorJusto*, da lugar a un prototipo* más completo *(en el sentido que tiene en cuenta más factores en su funcionamiento) que el que el obtenido inicialmente a partir del* SistemaDeSubastas*.*

En este ejemplo, en cambio, mostraremos cómo podemos lograr el mismo resultado por medio de la implementación de los métodos del subastador asociados a los enlaces que éste utiliza para comunicarse con los postores. Para ello, simplemente tendremos que declarar la variable n *en la especificación del componente* Subastador *(en vez de en el rol* Subastar*), e implementar los métodos que están relacionados con su manejo. Así, de forma similar a como especificamos en el* SubastadorJusto*, debemos incrementar* n *cada vez que se reciba una puja o un acuse de recibo uno de los postores (es decir, en los métodos* `pujaQUE` *y* `acuseQUE` *respectivamente), mientras que debemos inicializar* `n` *a cero cada vez que se difunda un nuevo precio a los postores (en el método* `precioBRD`*). Por último, debemos asociar la condición* `n == max` *a la acción interna que modela la decisión local de adjudicar o no el artículo:*

```
public Integer precioBRD(Integer valor)
    { n = 0; return valor; }
public void pujaQUE(Link respuesta)
    { n++; }
public void acuseQUE()
    { n++; }
public Boolean test211()
    { return new Boolean(n == max); }
```

Obsérvese que los métodos implementados tienen un argumento cuyo tipo depende del objeto transmitido durante la comunicación (así por ejemplo, `precioBRD` *tiene como argumento el valor del artículo). Los métodos asociados a acciones de entrada, como* `pujaQUE` *utilizan este argumento para permitir que el componente tenga conocimiento del valor leído en el enlace, y no devuelven ningún resultado. En cambio, los métodos asociados a acciones de salida, permiten al componente modificar este argumento antes de enviarlo por el enlace, por lo que devuelven dicho argumento como resultado. Por último, los métodos asociados a la comprobación de decisiones locales (como* `test211`*) no tienen argumentos, y devuelven un valor lógico que indica si el componente está dispuesto o no a realizar la acción, lo que es tenido en cuenta a la hora de establecer el compromiso del rol con una u otra acción de las que ofrece.*

Con esta implementación de los métodos del componente `Subastador` *logramos que éste ofrezca el mismo comportamiento que el presentado por* `SubastadorJusto`*, en el que el procedimiento seguido fue el de refinar el componente* `Subastador`*. La diferencia entre ambos enfoques reside en que en el primer caso las modificaciones están ocultas en la implementación del sistema, mientras que en éste último quedan reflejadas en la especificación en LEDA del mismo. Además, el proceso de refinamiento puede ser objeto de verificación formal utilizando la relación de extensión.*

Es probable que durante el desarrollo y pruebas del prototipo sea necesario realizar la generación de código varias veces, incluso después de haber comenzado a implementar los métodos del componente. En este proceso de generación se conserva la implementación de los métodos codificados por el usuario, de forma que no es necesaria su recodificación cada vez que se genera un nuevo prototipo.

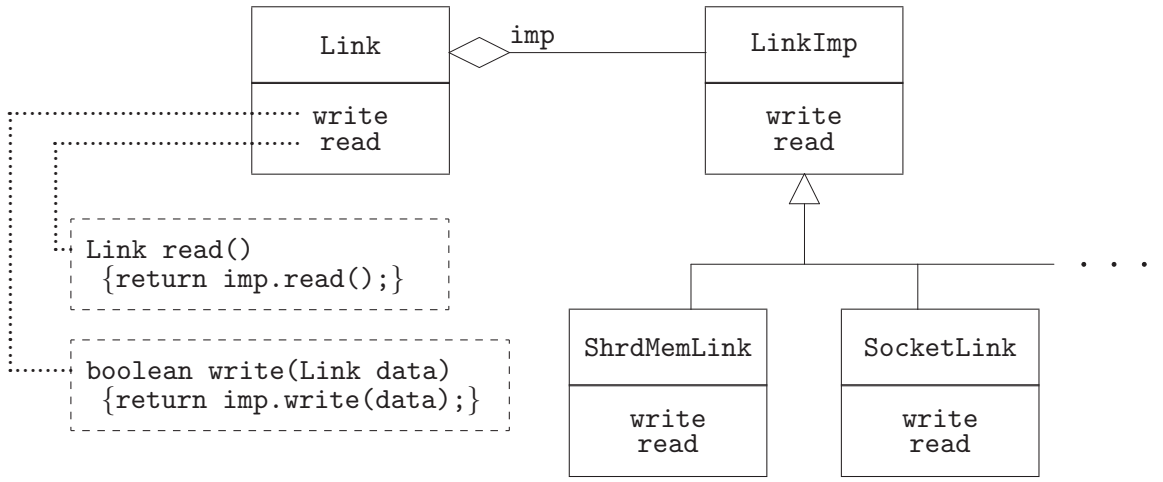


Figura 4.11: Implementación de diversos tipos de enlaces mediante el patrón *Puente*.

4.5.2 Comunicación entre roles

El prototipo del sistema descrito en las secciones anteriores consiste en un único proceso ejecutándose en una sola máquina, en la que residen todos los componentes del sistema. Cada rol se ejecuta en una hebra, y los enlaces entre roles implementan la comunicación de datos por medio de la compartición de variables que unos roles leen y otros escriben.

Esta implementación puede ser satisfactoria para las versiones iniciales del sistema, que se utilizan básicamente para probar la especificación y realizar pequeños ajustes sobre la misma, pero probablemente sea demasiado limitada para convertirse en el sistema final. Nuestro objetivo puede ser la construcción de un sistema distribuido, en el que posiblemente cada componente se ejecute en una máquina distinta, con lo que los enlaces ya no pueden basarse en el uso de memoria compartida, sino que utilizarán, por ejemplo, *sockets* o llamadas remotas a métodos (RMI) para comunicar entre sí los roles.

Esta variedad de mecanismos de comunicación queda encapsulada dentro de la clase `Link`, de forma que resulte transparente al diseñador y no influya en la implementación de los componentes ni de los roles. Para ello, hemos utilizado el patrón de diseño *Puente*, que permite desligar una abstracción de su implementación de forma que dispongamos de varias implementaciones alternativas de la abstracción.

La Figura 4.11 muestra la estructura del patrón *Puente* aplicado al diseño de los enlaces. La clase `Link` incorpora métodos para la lectura y escritura de los enlaces (`read` y `write`, respectivamente), métodos cuya implementación se delega a la clase `LinkImp`. Esta clase es abstracta, y de ella heredan las distintas implementaciones de los enlaces: `ShrdMemLink`, que implementa los enlaces utilizando memoria compartida (tal como hemos descrito en la Sección 4.2.3), mientras que `SocketLink` utiliza *sockets* para realizar esta implementación, siendo también posibles otras implementaciones alternativas. Cada una de estas subclases implementan los métodos `read` y `write` según el mecanismo de comunicación elegido. De acuerdo con esta estructura, los roles, que actúan como clientes de `Link` al leer y escribir los enlaces, no tienen conocimiento de las clases que finalmente realizan dichas lecturas y escrituras, por lo que es posible utilizar una implementación u otra, y obtener de esta forma prototipos que utilicen distintos mecanismos

de comunicación, sin que se vea afectada la especificación en LEDA ni el código de las clases generadas a partir de ella.

4.5.3 Coordinación de los roles de un componente

En las secciones anteriores hemos visto cuál es el proceso de desarrollo que se debe seguir una vez generado el prototipo a partir de la especificación del sistema. Este proceso transcurre a través de una serie de pasos o incrementos en los que se van implementando los aspectos computacionales de los componentes y es posible utilizar distintos mecanismos de comunicación entre ellos. Al final de este proceso dispondremos de un sistema cuyo comportamiento está regido por los patrones de interacción especificados en los roles, que son los encargados de invocar la funcionalidad ofrecida por los componentes.

Sin embargo, una implementación como la descrita plantea ciertas dificultades que deben ser salvadas para la obtención del sistema final, debido a que los roles pueden no contener toda la información sobre la coordinación de los componentes del sistema. Esto es debido a que son únicamente descripciones *parciales* de la interfaz de dichos componentes. Para ilustrar este punto consideremos el siguiente ejemplo.

Ejemplo 4.10 *Tomemos de nuevo el Ejemplo 2.12, en el que presentábamos un componente* Traductor*, cuya especificación en el cálculo* π *repetimos aquí:*

$$\begin{aligned} \mathit{Traductor}(e_1, e_2, s_1, s_2) = {} & e_1?(x).\ s_1!(x).\ \mathit{Traductor}(e_1, e_2, s_1, s_2) \\ & + e_2?(y).\ s_2!(y).\ \mathit{Traductor}(e_1, e_2, s_1, s_2) \end{aligned}$$

Como vemos, el Traductor *copia los datos recibidos por sus enlaces de entrada en sus enlaces de salida, de forma que realiza una suerte de traducción entre dichos nombres de enlaces.*

Supongamos ahora que hemos especificado dicho componente en LEDA utilizando dos roles, entrada *y* salida*, para describir su interfaz:*

```
component Traductor {
 interface
   entrada : Entrada(e1,e2) {
      e1?(x). Entrada(e1,e2)  +  e2?(y). Entrada(e1,e2)
   }
   salida : Salida(s1,s2) {
      t. (x)s1!(x). Salida(s1,s2)  +  t. (y)s2!(y). Salida(s1,s2)
   }
 }
```

El prototipo del sistema estaría formado por la traducción de los dos roles entrada *y* salida*, que invocan las operaciones* `e1QUE`*,* `e2QUE`*,* `s1EXC` *y* `s2EXC` *del componente* `Traductor` *siguiendo el protocolo indicado en la especificación de dichos roles en LEDA. En concreto, la presencia de una decisión local en el rol* `salida` *hará que la acción de salida* $s1!$ *se ejecute de forma independiente a si se ha producido o no inmediatamente antes la correspondiente entrada por* $e1$*, sucediendo lo mismo con las acciones a través de* $s2$ *y* $e2$*.*

Este funcionamiento independiente de los roles `entrada` *y* `salida` *puede bastar para un prototipo del sistema, pero no es admisible en la implementación del sistema final, en la que la*

acción de salida elegida habrá de depender de cuál haya sido la acción de entrada inmediatamente anterior. Para resolver esta dificultad, debemos implementar en el componente no sólo las acciones de entrada y salida por los enlaces, sino también los métodos de prueba de las acciones internas que configuran la citada decisión local en el rol `salida`*, de forma que el compromiso de la primera de las alternativas se realice tras la ejecución de un método* `test1()` *en el componente* `Traductor`*, mientras que la elección de la segunda alternativa vaya precedida de la invocación del método* `test2()`*. Podemos entonces establecer la ligadura de los roles* `entrada` *y* `salida` *del* `Traductor` *por medio de alguna variable que utilizaremos en estos métodos. La implementación de los métodos del componente podría ser la siguiente:*

```
void e1QUE(Object x) {
  // Procesamiento de la
       entrada recibida
    b = "e1";
  }

Boolean test1() {
  if ( b == "e1") {
    b = null;
    return Boolean.TRUE;
    }
  else return Boolean.FALSE;
  }
```

```
void e2QUE(Object y) {
  // Procesamiento de la
       entrada recibida
    b = "e2";
  }
Boolean test2() {
  if ( b == "e2") {
    b = null;
    return Boolean.TRUE;
    }
  else return Boolean.FALSE;
  }
```

Tal como vimos en la Sección 4.3, antes de decidir el compromiso del rol `salida` *con una de las acciones internas que figuran en las ramas de la alternativa no determinista, se evalúan los métodos de prueba* `test1()` *y* `test2()` *de forma que sólo se realizará el compromiso si el método de prueba correspondiente tiene como resultado el valor lógico verdadero. De esta forma la elección de una u otra alternativa podrá ser condicionada a cuál haya sido la operación de lectura ejecutada inmediatamente antes. En el prototipo inicial, resultado de la generación de código a partir de la especificación, todas los métodos de prueba sobre acciones internas devolverán como resultado el valor verdadero, estando de esta forma igualmente disponibles.*

Este ejemplo muestra cómo es posible utilizar los métodos del componente asociados a las acciones de los roles para controlar y coordinar el funcionamiento de dichos roles, de forma que la decisión de realizar unas acciones u otras dependa no sólo de la disponibilidad de dichas acciones, sino también del estado interno del componente. De esta manera conseguiremos dotar al sistema de la funcionalidad requerida, completando así su desarrollo.

4.6 Conclusiones

En este capítulo hemos esbozado un proceso de desarrollo de aplicaciones composicionales basado en el lenguaje LEDA. Nuestra propuesta combina las ventajas de utilizar un soporte formal para especificar la arquitectura del sistema, lo que permite la validación de ciertas propiedades del mismo, con un enfoque generativo e incremental que facilita el desarrollo de prototipos ejecutables y del propio sistema final.

El proceso de desarrollo está basado en las interfaces de los componentes del sistema, descritas por medio de una serie de roles. La arquitectura se especifica a través de conexiones entre roles, que describen cuáles son las relaciones que los componentes establecen unos con otros. A partir de esta especificación, el diseñador seguirá un proceso evolutivo de desarrollo, obteniendo desde el primer momento un prototipo que consiste básicamente en los protocolos de interacción descritos en los roles. Los prototipos generados están escritos en Java, aunque el enfoque propuesto es aplicable a cualquier otro lenguaje orientado a objetos.

El núcleo del prototipo lo constituyen las clases que implementan los conceptos fundamentales manejados en LEDA y, por extensión, en el cálculo π: componentes, roles, enlaces y acciones. El mecanismo distribuido de compromiso que hemos implementado determina las acciones a realizar en el sistema, normalmente consistentes en la coordinación de dos o más roles en la realización de acciones complementarias. El algoritmo de compromiso desarrollado es justo y no presenta problemas de bloqueo, aunque no es posible asegurar su convergencia. Este conjunto de clases básicas constituye un intérprete del cálculo π.

Por otra parte, el esquema de traducción presentado permite generar, a partir de la especificación de componentes y roles en LEDA, un conjunto de clases en Java que hacen uso de las citadas clases básicas para construir un prototipo ejecutable en el que se implementan los aspectos de coordinación del sistema especificado. Es importante destacar que el prototipo generado conserva las propiedades validadas sobre la especificación, y que su utilidad no consiste únicamente en la simulación del sistema, con objeto de probar o depurar la especificación, sino que sirve de base para la construcción del sistema final.

Así mismo, es posible utilizar diversos mecanismos de comunicación entre componentes, lo que da lugar a la generación de prototipos concurrentes, distribuidos, formados por uno o varios procesos, etc. El diseño del generador de código ha sido realizado de forma que esta variabilidad se refleje únicamente en las clases que representan los enlaces utilizados para conectar los roles, quedando totalmente oculta a la especificación en LEDA e incluso a las clases en Java generadas a partir de ella. Actualmente disponemos de una primera versión del generador de código, en el que se crea una hebra por cada uno de los roles, implementándose los enlaces entre roles mediante memoria compartida, por lo que el prototipo ejecutable generado consiste en un solo proceso. Así mismo, se encuentra en desarrollo una versión que utiliza comunicación mediante *sockets*, lo que permitirá obtener prototipos distribuidos, en los que los componentes estén asociados a procesos ejecutándose en ordenadores diferentes conectados a través de Internet.

Al margen de la posibilidad de obtener prototipos ejecutables a partir de la especificación en LEDA, un valor añadido del esquema de generación propuesto consiste en el proceso iterativo de desarrollo que se deriva de él, que proporciona una guía para pasar de una especificación en LEDA a una implementación del sistema en Java. De este modo, una vez obtenido el prototipo inicial, el desarrollo del sistema final se lleva a cabo mediante un proceso evolutivo e incremental, en el que se completan los aspectos computacionales de los componentes, dotando de esta forma al sistema de la funcionalidad adecuada. Durante este proceso se pueden corregir errores en la especificación, refinarla por medio de la extensión de componentes y roles, y proceder repetidas veces a su análisis y a la generación de diversos prototipos a partir de ella.

Por último, es necesario señalar que, si bien el prototipo inicial del sistema presenta las mismas propiedades que fueron verificadas sobre la especificación, el proceso de desarrollo seguido para obtener el sistema final a partir del prototipo no permite asegurar que se preserven dichas propiedades. En efecto, al implementar la funcionalidad de los componentes, el programador puede modificar el comportamiento de los mismos, interfiriendo de esta manera en el protocolo de coordinación establecido por los roles.

Capítulo 5

Conclusiones y trabajo futuro

Un trabajo como el que presentamos aquí no puede considerarse nunca como cerrado, sino que siempre es susceptible de continuación y ampliación. De hecho, el presente trabajo continúa evolucionando a medida que surgen nuevas implicaciones y ramificaciones del mismo, y a medida que aportaciones de otros autores puedan ser incorporadas a su desarrollo. No obstante, ha llegado el momento de poner punto final a esta memoria. En este último capítulo se pretende hacer balance del trabajo desarrollado, sus aportaciones y sus limitaciones, así como apuntar diversas líneas de continuación del mismo.

5.1 Conclusiones

Los aspectos arquitectónicos del diseño de software han ido cobrando interés a medida que aumenta la complejidad de las aplicaciones. En los últimos años, el término Arquitectura del Software ha sido acuñado para hacer referencia a la disciplina que, como parte de la Ingeniería del Software, se ocupa de todo lo relacionado con este nivel del proceso de desarrollo. Se trata de un campo relativamente reciente, en el que se están cosechando los primeros resultados, pero en donde sin duda queda aún mucho por hacer.

Uno de los objetivos fundamentales de la Arquitectura del Software es servir de ayuda a la toma de decisiones —en las etapas iniciales del desarrollo— respecto a la estructura de las aplicaciones. Para ello, es necesario anticipar las consecuencias de estas decisiones, es decir, cuáles van a ser las propiedades que, como resultado de una u otra decisión, presente el sistema una vez implementado. Únicamente desde el rigor que proporciona un sólido soporte formal es posible analizar y descubrir por anticipado el impacto que una u otra decisión tendrá en el producto resultante. Por tanto, una primera conclusión de este trabajo sería la necesidad, o al menos la utilidad y la conveniencia, de usar métodos formales en Arquitectura del Software.

Ahora bien, el éxito en la aplicación de métodos formales al desarrollo de software dependerá de nuestra habilidad para construir modelos y formalismos adecuados a cada una de las etapas del proceso de desarrollo. En este sentido, nuestro trabajo demuestra la adecuación del cálculo π para la descripción arquitectónica, debido fundamentalmente a la capacidad de expresar de forma natural la movilidad en este álgebra de procesos, lo que permite la descripción de arquitecturas dinámicas, cuya configuración pueda ir evolucionando a lo largo del tiempo.

Sin embargo, una consecuencia inmediata de la aplicación del cálculo π a la descripción del comportamiento del software es que el bajo nivel de esta notación complica notablemente la especificación de sistemas complejos. Llegamos, por tanto, a la conclusión de que es necesario

desarrollar notaciones de más alto nivel, que oculten al usuario en la medida de lo posible las dificultades asociadas al uso del formalismo elegido. Por este motivo, hemos desarrollado LEDA, un lenguaje para la descripción de arquitecturas de software dinámicas basado en el cálculo π. Una especificación en LEDA describe un sistema de software como un conjunto de componentes interconectados. Las especificaciones son jerárquicas y composicionales, de forma que un componente puede construirse a partir de la interconexión de otros más sencillos. El comportamiento de cada componente se describe mediante los *roles* o papeles abstractos que el componente juega en el sistema. Cada rol especifica parcialmente el comportamiento de un componente por medio del protocolo de interacción que éste sigue en su funcionamiento. Este protocolo regirá sus relaciones con el resto del sistema del que forma parte.

A pesar de la juventud de la Arquitectura del Software, ya se ha propuesto toda una serie de lenguajes de descripción arquitectónica. Al comienzo de esta memoria hemos descrito brevemente las características de los más relevantes de entre ellos. Como consecuencia de este estudio, hemos determinado algunos de los requisitos que debe satisfacer un ADL, haciendo especial hincapié en aquellas características que no han sido abordadas, o al menos no de forma adecuada, en esta primera generación de lenguajes. La evaluación del cumplimiento de estos requisitos por parte de los diferentes ADLs estudiados dio lugar a una tabla comparativa que ahora reproducimos aquí pero incluyendo además a LEDA entre los lenguajes objeto de comparación.

Entre las principales aportaciones de LEDA, que lo distinguen de otros lenguajes dentro de esta categoría, destaca en primer lugar su capacidad de describir arquitecturas dinámicas, de topología cambiante, en las que las relaciones entre componentes no son rígidas, sino que se modifican durante la ejecución del sistema. La mayoría de las propuestas realizadas hasta la

	Entidades	Dinamismo	Verificación de propiedades	Desarrollo y reutilización	Ejecución
UniCon	Componentes y conectores	No	No	No	Generación de código
Wright	Componentes y conectores	No	Compatibilidad	No	No
Darwin	Componentes	Sí	Seguridad y viveza	No	Generación de código
Rapide	Componentes	Limitado	Análisis de restricciones	Herencia	Simulación y generación
C2	Componentes	Limitado	No	Subtipado	Generación de código
LEDA	**Componentes**	**Sí**	**Compatibilidad y extensión**	**Extensión y genericidad**	**Simulación y generación**

Figura 5.1: Tabla comparativa de las características de LEDA y los ADLs estudiados.

fecha en este campo sólo permiten describir arquitecturas estáticas, en las que los componentes que las forman y las relaciones entre éstos permanecen inmutables. La descripción de estructuras dinámicas y cambiantes es un campo de interés que aún se está explorando.

En segundo lugar, debemos hacer mención del mecanismo de herencia que ofrece el lenguaje, tanto a la hora de especificar los propios componentes como las interacciones entre ellos. Se trata ésta de una característica novedosa dentro de los lenguajes de descripción de arquitecturas, y que permite dotar a LEDA de un mecanismo de extensión y adaptación al cambio similar al que tan buenos resultados ha dado en el contexto de los lenguajes orientados a objetos. La extensión no sólo hace posible la clasificación y el establecimiento de relaciones de parentesco entre componentes y entre interacciones, algo que es muy necesario para organizar un campo de estudio aún incipiente, sino que sirve además como medio para el refinamiento de las especificaciones, en sucesivos niveles de detalle, y para su adaptación a cambios en los requisitos.

Otra de las aportaciones del lenguaje es la definición de arquitecturas genéricas, que pueden ser parametrizadas posteriormente dando lugar a la especificación de sistemas distintos. Todos los componentes de una arquitectura en LEDA actúan como parámetros formales que pueden ser instanciados con componentes concretos. Consideramos esta característica especialmente interesante, puesto que permite elevar la expresividad de nuestro lenguaje de la descripción de la arquitectura de sistemas individuales a la de patrones arquitectónicos. De este modo, las arquitecturas descritas en LEDA pueden ser consideradas como marcos de trabajo genéricos, que admiten ser instanciados y reutilizados tantas veces como sea necesario. Se consigue así la reutilización, no sólo de los componentes, sino del propio diseño arquitectónico, que es reutilizado en las aplicaciones construidas sobre el marco de trabajo.

Nuestra propuesta se apoya en un álgebra de procesos —el cálculo π— como soporte formal para la descripción y análisis de las arquitecturas. Por el contrario, una parte importante de las notaciones propuestas carece de un fundamento formal adecuado, lo que limita su uso a lo meramente descriptivo.

En nuestro caso la base formal del lenguaje permite, en primer lugar, la simulación de los sistemas especificados. En efecto, hemos expresado la semántica de LEDA en términos del cálculo π, de manera que es posible realizar una traducción de las especificaciones, que pueden a continuación ser simuladas utilizando para ello un intérprete del cálculo.

En segundo lugar, el uso de una notación formal para la descripción del comportamiento de los componentes ofrece interesantes posibilidades de análisis. Con este objeto, se han definido relaciones de compatibilidad, herencia y extensión de comportamiento en el contexto del cálculo π. Como hemos visto, estas relaciones permiten la verificación automática de propiedades de composicionalidad y reemplazabilidad de las arquitecturas, en particular para determinar si el comportamiento de los componentes conectados es compatible, o si un comportamiento puede considerarse como una especialización o extensión de otro. Otra característica interesante del lenguaje es la posibilidad de definir en LEDA adaptadores, con el objeto de poder combinar componentes que no sean compatibles.

Por último, nuestra propuesta no se limita sólo a la especificación y análisis de arquitecturas, sino que se completa con una guía para el proceso de desarrollo a partir de la descripción arquitectónica. El reto consistía aquí en obtener una implementación fiel a la especificación original, en la que se siguiesen satisfaciendo las propiedades demostradas durante el análisis. Para ello, a partir de la especificación en LEDA se genera un prototipo que sirve de base para un proceso de desarrollo iterativo e incremental. El lenguaje elegido para la generación de código es Java, aunque nuestra propuesta es válida para cualquier otro lenguaje orientado a objetos.

5.2 Trabajo futuro

En esta sección se apuntan brevemente algunas líneas de trabajo que consideramos de especial interés para la continuación del trabajo presentado en esta memoria.

En primer lugar, los resultados presentados en el Capítulo 2, referentes al análisis de propiedades en LEDA, sugieren de forma inmediata una extensión, consistente en aprovechar el soporte formal que proporciona el cálculo π para la verificación de otras propiedades, aparte de las ya definidas de compatibilidad y herencia de comportamiento. Resulta especialmente atractivo el trabajo desarrollado en este sentido sobre el lenguaje Darwin [Giannakopoulou et al., 1999] —al que ya hemos hecho referencia en la Sección 2.5, y donde se especifica el comportamiento de los componentes con objeto verificar su compatibilidad. En dicho trabajo se argumenta además la posibilidad de analizar otras propiedades de seguridad y viveza, siempre que dichas propiedades puedan especificarse de forma adecuada mediante máquinas de estado finito. Consideramos que esta idea puede ser aplicada de forma análoga a nuestro entorno, de manera que si especificásemos como agentes en el cálculo π, tanto un componente (sus roles, para ser más exactos), como una cierta propiedad que este componente deba mostrar, sería posible comprobar por medio de un análisis de compatibilidad entre dichos agentes, si el comportamiento del componente satisface la propiedad.

Siguiendo en la misma línea de extensión de la capacidad expresiva y de análisis de nuestra propuesta, tanto el estudio realizado sobre los diferentes ADLs como nuestro propio trabajo muestran cómo expresar las propiedades estructurales de los sistemas de software –y también cómo el uso de notaciones formales en los ADLs hace posible razonar sobre dichas propiedades. No obstante, existen numerosos aspectos muy importantes en cualquier aplicación, como son los no funcionales, que no es posible expresar directamente en términos de la estructura del sistema. Por consiguiente, otra interesante línea de trabajo futuro consiste en determinar cómo extender LEDA para expresar estas propiedades y de qué forma es posible proceder a su verificación. Ya en los trabajos iniciales sobre Arquitectura del Software se apuntaba la posibilidad de que el nivel arquitectónico fuese el adecuado para especificar y analizar estas propiedades no funcionales [Shaw y Garlan, 1995], citándose entre otros los aspectos de rendimiento, fiabilidad, seguridad, respuesta temporal, precisión, etc. Aunque existen algunos interesantes trabajos sobre la descripción de propiedades no funcionales de las arquitecturas de software [Franch, 1997, Franch y Botella, 1998], la impresión generalizada es que los avances en este campo no han producido aún resultados significativos [Shaw, 1999, Szyperski, 2000], debido en gran medida a la enorme disparidad de las propiedades no funcionales a considerar. En este sentido, parece razonable asumir que no existe un único formalismo en el que se pueda representar y analizar toda la diversidad de propiedades de interés en el campo de la Arquitectura del Software, lo que implica la necesidad de combinar el formalismo base de LEDA —el cálculo π— con otros, de manera que sea posible tratar un número significativo de propiedades.

Continuando con esta descripción de líneas de trabajo futuro, debemos comentar que la relación de extensión que hemos presentado en el Capítulo 2 admite la posibilidad de herencia múltiple, en el sentido de que un rol puede refinar o extender no a un solo progenitor, sino a varios. No obstante, esta posibilidad no ha sido explotada en el lenguaje LEDA, presentado en el Capítulo 3, con objeto de evitar —al menos en esta primera versión del lenguaje— los conflictos de herencia repetida o las restricciones sobre la herencia múltiple que aparecen habitualmente en los lenguajes orientados a objetos que presentan esta posibilidad. Una línea de trabajo futuro consiste pues en abordar la herencia múltiple en LEDA, su utilidad y la problemática que pueda plantear.

Otra interesante línea de trabajo futuro consiste en profundizar en el estudio de la herencia y derivación de roles como condición que asegure la reemplazabilidad de unos componentes por otros. Esta línea de trabajo puede desarrollarse siguiendo al menos dos enfoques complementarios. En primer lugar, considerando las relaciones que se establecen entre la descripción del comportamiento de roles o componentes representados a distintos niveles de abstracción, como proponen diversos trabajos ya comentados en la Sección 2.5, en los que una acción atómica de comunicación da lugar a toda una serie de acciones —es decir, a un proceso— si disminuimos el nivel de abstracción [Aceto y Hennessy, 1994, Gorrieri y Rensink, 2000]. En segundo lugar, considerando la relación de herencia como garante de la reemplazabilidad *en un contexto determinado*, para lo cual los trabajos existentes relativos a la definición de una relación de equivalencia dependiente de contexto [Larsen, 1987] son un interesante punto de partida. Esto nos permitiría relajar las restricciones impuestas por la relación de la herencia presentada en el Capítulo 2, restricciones debidas fundamentalmente a que en su presente caracterización, la herencia asegura la reemplazabilidad de un componente por otro *en cualquier contexto*.

Dejando ya a un lado los aspectos formales de nuestra propuesta, otra línea de trabajo futuro consiste en la construcción de toda una gama de herramientas que faciliten la especificación utilizando LEDA, el análisis de las arquitecturas descritas y el proceso de desarrollo del sistema final a partir de la especificación. A este respecto, disponemos en la actualidad de herramientas para el análisis de la compatibilidad entre componentes, implementadas a partir de MWB [Victor, 1994], y de una primera versión del generador de código que utiliza memoria compartida para comunicar entre sí los componentes, estando en desarrollo una versión que implementa dicha comunicación mediante *sockets*. Esto último permitirá generar prototipos distribuidos, en los que los componentes estén asociados a procesos ejecutándose en ordenadores diferentes conectados a través de Internet.

Del mismo modo, sería especialmente interesante el desarrollo de herramientas para el análisis de las relaciones de herencia y extensión de comportamiento de agentes especificados en el cálculo π, lo que permitiría la comprobación de las declaraciones de extensión, tanto de roles como de componentes, en el lenguaje LEDA, y de la instanciación de arquitecturas. Otras posibles herramientas incluyen editores gráficos que faciliten la utilización del lenguaje de especificación, así como herramientas CASE para simplificar el proceso de implementación del sistema final.

A este respecto debemos señalar que la construcción de herramientas de carácter *industrial* requiere un considerable esfuerzo que no puede ser afrontado de forma individual. En este sentido, una propuesta interesante es ACME [Garlan et al., 1997], que ha sido definido como lenguaje de intercambio entre los diferentes ADLs, de manera que facilite la difusión de las herramientas construidas para cada uno de ellos. El desarrollo de traductores entre LEDA y ACME permitiría la integración de dichas herramientas en nuestro entorno, al menos en la medida en que los diferentes enfoques y formalismos utilizados en los diversos ADLs lo hiciesen posible.

Por último, señalar una línea que más bien debemos considerar como trabajo *presente* —debido a que ya ha comenzado a dar sus frutos—, y que consiste en la aplicación de las principales aportaciones de este trabajo a otros modelos y notaciones utilizados en Ingeniería del Software. En este sentido, no podemos pretender que LEDA se convierta en el lenguaje estándar para la descripción arquitectónica, sino que su desarrollo nos ha permitido conocer el estado del arte en este campo y adquirir experiencia sobre cuáles son los problemas actualmente existentes y sus posibles soluciones. Esta experiencia puede ser aplicada ahora en otros contextos y a otros lenguajes. Actualmente estamos trabajando en la extensión de los lenguajes de descripción de interfaz (IDLs) de forma que no sólo indiquen cuáles son los servicios que ofrece un componente,

sino también los que precisa para su funcionamiento, y cuál es el protocolo que debe seguirse para utilizarlo [Canal et al., 1999b]. También en esta línea, hemos propuesto una extensión del IDL de CORBA, mediante el uso de una notación basada en el cálculo π, que incorpora las nociones de rol, compatibilidad y herencia de comportamiento que hemos presentado en esta memoria [Canal et al., 2000].

Apéndice A

Sintaxis BNF

A continuación presentamos la gramática del lenguaje LEDA utilizando para ello notación BNF:

<*arquitectura*>$_1$ ::= <*instancia*>$_2$ |
<*declaración de clase*>$_3$ <*arquitectura*>$_1$

<*instancia*>$_2$ ::= **instance** <*nombre*>$_{69}$ ':'
<*identificador de clase de componentes*>$_{35}$

<*declaración de clase*>$_3$ ::= **role** <*declaración de clase de roles*>$_4$ |
component <*declaración de clase de componentes*>$_{33}$

<*declaración de clase de roles*>$_4$ ::= <*definición de agente*>$_{24}$
<*declaración de extensión de rol*>$_5$
<*cuerpo de declaración de clase de roles*>$_7$

<*declaración de extensión de rol*>$_5$ ::= ε |
extends <*identificador de clase de roles*>$_6$

<*identificador de clase de roles*>$_6$ ::=
<*nombre*>$_{69}$ |
<*identificador de clase de roles*>$_6$ '.' <*nombre*>$_{69}$

<*cuerpo de declaración de clase de roles*>$_7$ ::= '{'
<*declaración de constantes*>$_8$
<*declaración de variables*>$_9$
<*especificación de rol*>$_{15}$
'}'

<*declaración de constantes*>$_8$ ::= ε |
constant <*lista de nombres*>$_{68}$ ';'

<*declaración de variables*>$_9$ ::= ε |
var <*lista de declaración de variables*>$_{10}$ ';'

<*lista de declaración de variables*>$_{10}$ ::=
<*declaración de grupo de variables*>$_{11}$ |
<*declaración de grupo de variables*>$_{11}$ ';' <*lista de declaración de variables*>$_{10}$

<*declaración de grupo de variables*>$_{11}$ ::=
<*lista de nombres de vectores*>$_{12}$ ':'
<*identificador de tipo de datos*>$_{59}$ <*valor inicial*>$_{60}$

<*lista de nombres de vectores*>$_{12}$ ::=
<*nombre de vector*>$_{13}$ |
<*nombre de vector*>$_{13}$ ',' <*lista de nombres de vectores*>$_{12}$

<*nombre de vector*>$_{13}$ ::= <*nombre*>$_{69}$ |
<*nombre*>$_{69}$ '[' <*índice de vector*>$_{14}$ ']'

<*índice de vector*>$_{14}$ ::= ε |
'*' |
<*constante positiva de tipo entero*>$_{64}$

<*especificación de rol*>$_{15}$ ::= <*especificación de agentes*>$_{16}$ |
<*especificación de extensión de rol*>$_{20}$
<*lista de agentes*>$_{18}$

<*especificación de agentes*>$_{16}$ ::= **spec is** <*especificación de comportamiento*>$_{17}$ ';'
<*lista de agentes*>$_{18}$

<*especificación de comportamiento*>$_{17}$ ::= **implicit** |
<*especificación en cálculo* π>$_{25}$

<*lista de agentes*>$_{18}$ ::= ε |
<*especificación de agente*>$_{19}$ ';' <*lista de agentes*>$_{18}$

<*especificación de agente*>$_{19}$ ::= **agent** <*definición de agente*>$_{24}$ **is**
<*especificación en cálculo* π>$_{25}$

<*especificación de extensión de rol*>$_{20}$ ::=
<*elemento de extensión de rol*>$_{21}$ |
<*elemento de extensión de rol*>$_{21}$ <*especificación de extensión de rol*>$_{20}$

<*elemento de extensión de rol*>$_{21}$ ::= <*redefinición de agente*>$_{22}$ |
<*adición de agente*>$_{23}$

<*redefinición de agente*>$_{22}$ ::= **redefining** <*nombre*>$_{69}$
as <*especificación en cálculo* π>$_{25}$ ';'

<*adición de agente*>$_{23}$ ::= **redefining** <*nombre*>$_{69}$
adding <*especificación en cálculo* π>$_{25}$ ';'

<*definición de agente*>$_{24}$ ::= <*nombre*>$_{69}$ '(' <*lista de nombres*>$_{68}$ ')'

<*especificación en cálculo* π>$_{25}$ ::= <*agente*>$_{26}$

<*agente*>$_{26}$ ::= **0** |
'(' <*agente*>$_{26}$ ')' |
'(' <*lista de nombres*>$_{68}$ ')' <*agente*>$_{26}$ |
'(' **new** <*nombre*>$_{69}$ ')' <*agente*>$_{26}$ |
'[' <*condición*>$_{27}$ ']' <*agente*>$_{26}$ |
<*acción prefija*>$_{29}$ '.' <*agente*>$_{26}$ |
<*agente*>$_{26}$ '|' <*agente*>$_{26}$ |
<*agente*>$_{26}$ '+' <*agente*>$_{26}$ |
<*definición de agente*>$_{24}$

<*condición*>$_{27}$ ::= '(' <*condición*>$_{27}$ ')' |
not <*condición*>$_{27}$ |
<*condición*>$_{27}$ **or** <*condición*>$_{27}$ |
<*condición*>$_{27}$ **and** <*condición*>$_{27}$ |
<*término*>$_{28}$ '=' <*término*>$_{28}$ |
<*término*>$_{28}$ '>' <*término*>$_{28}$ |
<*término*>$_{28}$ '<' <*término*>$_{28}$ |
<*término*>$_{28}$ '>=' <*término*>$_{28}$ |
<*término*>$_{28}$ '<=' <*término*>$_{28}$

<*término*>$_{28}$ ::= '(' <*término*>$_{28}$ ')' |
'-' <*término*>$_{28}$ |
<*término*>$_{28}$ '+' <*término*>$_{28}$ |
<*término*>$_{28}$ '-' <*término*>$_{28}$ |
<*término*>$_{28}$ '*' <*término*>$_{28}$ |
<*término*>$_{28}$ '/' <*término*>$_{28}$

<*acción prefija*>$_{29}$ ::= <*acción de comunicación*>$_{30}$ |
<*acción interna*>$_{32}$

<*acción de comunicación*>$_{30}$ ::= <*nombre*>$_{69}$ '!(' <*lista de objetos*>$_{31}$ ')' |
<*nombre*>$_{69}$ '[*]!(' <*lista de objetos*>$_{31}$ ')' |
<*nombre*>$_{69}$ '?(' <*lista de objetos*>$_{31}$ ')' |
<*nombre*>$_{69}$ '[ ]?(' <*lista de objetos*>$_{31}$ ')'

<*lista de objetos*>$_{31}$ ::= ε |
<*lista de nombres*>$_{68}$

<*acción interna*>$_{32}$ ::= **t** |
<*nombre*>$_{69}$ ':=' <*término*>$_{28}$ |
<*nombre*>$_{69}$ '++' |
<*nombre*>$_{69}$ '- -'

<*declaración de clase de componentes*>$_{33}$ ::=
<*nombre*>$_{69}$
<*declaración de extensión de componente*>$_{34}$
<*cuerpo de declaración de clase de componentes*>$_{39}$

<*declaración de extensión de componente*>$_{34}$ ::=
ε |
extends <*identificador de clase de componentes*>$_{35}$

<*identificador de clase de componentes*>$_{35}$ ::=
<*nombre*>$_{69}$ |
<*nombre*>$_{69}$ '[' <*lista de declaración de componentes*>$_{36}$ ']'

<*lista de declaración de componentes*>$_{36}$ ::=
<*declaración de grupo de componentes*>$_{37}$ |
<*declaración de grupo de componentes*>$_{37}$
<*lista de declaración de componentes*>$_{36}$

<*declaración de grupo de componentes*>$_{37}$ ::=
<*lista de nombres de vectores*>$_{12}$
<*identificación de clase de componentes o adaptadores*>$_{38}$

<*identificación de clase de componentes o adaptadores*>$_{38}$ ::=
':' **any** ';' |
':' <*identificador de clase de componentes*>$_{35}$ ';'
<*identificación de clase de roles*>$_{44}$

<*cuerpo de declaración de clase de componentes*>$_{39}$ ::= '{'
<*declaración de constantes*>$_{8}$
<*declaración de variables*>$_{9}$
<*declaración de interfaz*>$_{40}$
<*declaración de composición*>$_{45}$
<*declaración de conexiones*>$_{46}$
'}'

<*declaración de interfaz*>$_{40}$ ::= **interface** <*descripción de interfaz*>$_{41}$ ';'

<*descripción de interfaz*>$_{41}$ ::= **none** |
<*lista de declaración de roles*>$_{42}$

<*lista de declaración de roles*>$_{42}$ ::=
<*declaración de grupo de roles*>$_{43}$ |
<*declaración de grupo de roles*>$_{43}$ ';' <*lista de declaración de roles*>$_{42}$

<*declaración de grupo de roles*>$_{43}$ ::= <*lista de nombres de vectores*>$_{12}$
<*identificación de clase de roles*>$_{44}$

<*identificación de clase de roles*>$_{44}$ ::=
':' <*identificador de clase de roles*>$_{6}$ ';' |
':' <*declaración de clase de roles*>$_{4}$ |
<*cuerpo de declaración de clase de roles*>$_{7}$

<*declaración de composición*>$_{45}$ ::=
ε |
composition <*lista de declaración de componentes*>$_{36}$

<*declaración de conexiones*>$_{46}$ ::= ε |
attachments <*lista de conexiones*>$_{47}$ ';'

<*lista de conexiones*>$_{47}$ ::= <*conexión*>$_{48}$ |
<*conexión*>$_{48}$ ';' <*lista de conexiones*>$_{47}$

<*conexión*>$_{48}$ ::= <*conexión interna*>$_{49}$ |
<*conexión de exportación*>$_{57}$

<*conexión interna*>$_{49}$ ::=
<*término de conexión*>$_{50}$ '<>' <*término de conexión*>$_{50}$ |
<*término de conexión*>$_{50}$ '<>' <*conexión interna*>$_{49}$

<término de conexión>$_{50}$ ::= <término de conexión incondicional>$_{51}$ |
<término de conexión condicional>$_{53}$ |
<término de conexión condicional múltiple>$_{54}$

<término de conexión incondicional>$_{51}$ ::=
<identificación de instancia>$_{52}$ '(' <lista de nombres>$_{68}$ ')'

<identificación de instancia>$_{52}$ ::=
<nombre de vector>$_{13}$ |
<nombre de vector>$_{13}$ '.' <identificación de instancia>$_{52}$

<término de conexión condicional>$_{53}$ ::=
if <condición>$_{27}$
then <término de conexión incondicional>$_{51}$
else <término de conexión incondicional>$_{51}$

<término de conexión condicional múltiple>$_{54}$ ::=
<lista de alternativas de conexión>$_{55}$
default <término de conexión incondicional>$_{51}$

<lista de alternativas de conexión>$_{55}$ ::=
<alternativa de conexión>$_{56}$ |
<alternativa de conexión>$_{56}$ <lista de alternativas de conexión>$_{55}$

<alternativa de conexión>$_{56}$ ::= **case** <condición>$_{27}$
then <término de conexión incondicional>$_{51}$

<conexión de exportación>$_{57}$ ::=
<término izquierdo de exportación>$_{61}$ '>>' <término de conexión incondicional>$_{51}$

<término izquierdo de exportación>$_{58}$ ::=
<término de conexión>$_{50}$ |
<término de conexión>$_{50}$ ',' <término izquierdo de exportación>$_{58}$

<identificador de tipo de datos>$_{59}$ ::= **Integer** |
Float |
Character |
Boolean

<valor inicial>$_{60}$ ::= ε |
':=' <constante de tipo>$_{61}$

<*constante de tipo*>$_{61}$ ::= <*constante de tipo entero*>$_{62}$ |
<*constante de tipo decimal*>$_{65}$ |
<*constante de tipo carácter*>$_{66}$ |
<*constante de tipo booleano*>$_{67}$

<*constante de tipo entero*>$_{62}$ ::= <*signo*>$_{63}$ <*constante positiva de tipo entero*>$_{64}$

<*signo*>$_{63}$ ::= ε | '+' | '-'

<*constante positiva de tipo entero*>$_{64}$ ::=
<*dígito*>$_{73}$ |
<*dígito*>$_{73}$ <*constante positiva de tipo entero*>$_{64}$

<*constante de tipo decimal*>$_{65}$ ::=
<*constante de tipo entero*>$_{62}$ |
<*constante de tipo entero*>$_{62}$ '.' <*constante positiva de tipo entero*>$_{64}$

<*constante de tipo carácter*>$_{66}$ ::= "' <*carácter*>$_{71}$ "'

<*constante de tipo booleano*>$_{67}$ ::= **true** | **false**

<*lista de nombres*>$_{68}$ ::= <*nombre*>$_{69}$ |
<*nombre*>$_{69}$ ',' <*lista de nombres*>$_{68}$

<*nombre*>$_{69}$ ::= <*letra*>$_{72}$ <*nombre alfanumérico*>$_{70}$

<*nombre alfanumérico*>$_{70}$ ::= ε |
<*carácter*>$_{71}$ <*nombre alfanumérico*>$_{70}$

<*carácter*>$_{71}$ ::= <*letra*>$_{72}$ | <*dígito*>$_{73}$

<*letra*>$_{72}$ ::= **a** | ... | **z** | **A** | ... | **Z**

<*dígito*>$_{73}$ ::= **0** | ... | **9**

Apéndice B

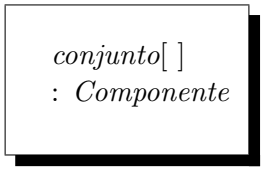
Notación gráfica

Componentes

Clase de componentes

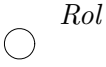
Instancia de componente

Conjunto de instancias

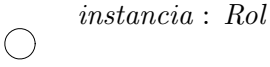


Roles

Clase de roles



Instancia de rol

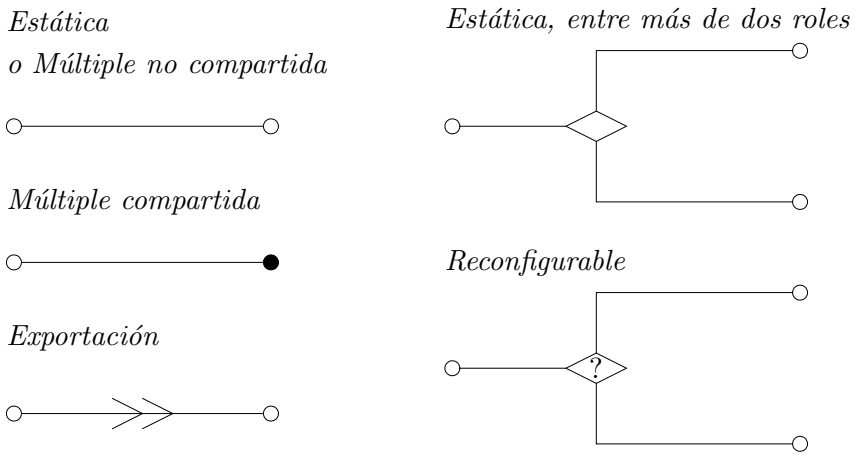


Adaptadores

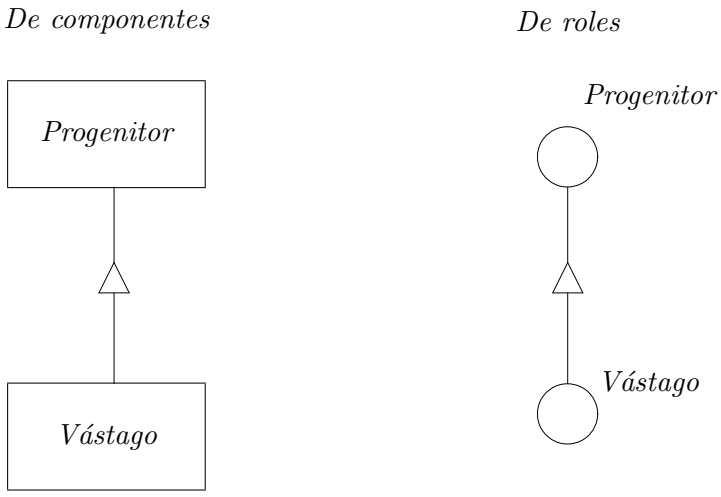
Clase de adaptadores

Instancia de adaptador

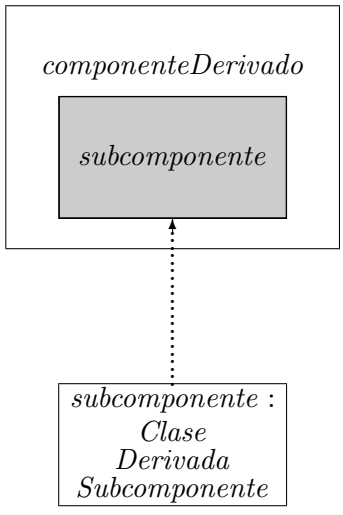
Conexiones



Extensión



Instanciación



Apéndice C

Prototipo del Sistema de Subastas

C.1 Clases básicas

Clase `Component`

```
abstract class Component {

    String name;
    int pid;

  public Component(String name, int pid) {
    this.name = name;
    this.pid = pid;
    if ( pid != 0 )
      this.name = this.name + "[" + pid + "]";
    System.out.println("(new " + this.name + ")");
    }

  public String toString() {
    return name;
    }

  abstract public Object clone();

  abstract public void attach(Link links[]);

  public Object get(String field) {
    Object value = null;
    try {
      value = getClass().getField(field).get(this);
      }
    catch (Exception e) {
      System.out.println("Field " + name + "." + field + " doesn't exist.");
      }
    return value;
    }
```

```
public void set(String field, Object value) {
  try {
    getClass().getField(field).set(this,value);
    }
  catch (Exception e) {
    System.out.println("Field " + name + "." + field + " doesn't exist.");
    }
  }

public Object perform(String method, Object prmtr) {
  try {
    Class formal[];
    Object real[];
    if ( prmtr == null ) {
      formal = null;
      real = null;
      }
    else {
     formal = new Class[1];
     formal[0] = prmtr.getClass();
     real = new Object[1];
     real[0] = prmtr;
     }
    return this.getClass().getMethod(method,formal).invoke(this,real);
    }
  catch (Exception e) {
    System.out.println("Method " + name + "." + method + "(" + prmtr + ")");
    return prmtr;
    }
  }

public boolean test(Tau action) {
  Object test = perform("test" + action.number, null);
  if ( test != null )
    return ((Boolean) test).booleanValue();
  else
    return true; // default
  }
}
```

Clase `Role`

```
class Role implements Runnable {

    String name;
    int pid;
    Thread t;
    Component component;
    State state;
    ActionList offers;
    Action commitment;

  public Role(String name, int pid, Component component) {
    this.name = name;
    this.pid = pid;
    if ( pid != 0 )
      this.name = this.name + "[" + pid + "]";
    t = new Thread(this,this.name);
    this.component = component;
    state = null;
    offers = new ActionList();
    commitment = null;
    System.out.println("(new " + this.name + ")");
    }

  public String toString() {
    return name;
    }

  public void attach(Link links[]) {
    for(int i=0; i<links.length; i++)
      links[i].incr();
    state.attach(links);
    t = new Thread(this, name);
    t.start();
    }

  public void run() {
    while ( state != null )
      state = state.spec();
    }

  public synchronized boolean setCommitment(Action a) {
    if ( commitment == null ) {
      commitment = a;
      return true;
      }
    else return false;
    }
```

```
public synchronized void resetCommitment() {
  commitment = null;
  }

public synchronized Action getCommitment() {
  return commitment;
  }

int select() {
  resetCommitment();
  while ( getCommitment() == null ) {
    t.yield();
    tryCommitment();
    }
  removeOffers();
  return commitment.number;
  }

void tryCommitment() {
  Action a = offers.atRandom();
  if ( a != null ) {
    if ( a.sign == 0 ) {
      // TAU action
      Tau action = (Tau) a;
      // Tests whether to commit or not to the action
      if ( component.test(action) ) {
        setCommitment(a);
        }
      }
    else {
      // INPUT/OUTPUT action
      IOAction action = (IOAction) a;
      IOAction notAction = (IOAction)(action.subject.offers.find(-action.sign));
      if (notAction != null) {
        Role q = notAction.role;
        if ( (q!= this) ) {
          // Tries to commit both roles
          Lock.commit(this,action,q,notAction);
          }
        }
      }
    }
  }

Object perform() {
  return commitment.perform();
  }
```

```
  void removeOffers() {
    while ( offers.action != null ) {
      Action a = offers.action;
      offers.remove(a);
      if ( a.sign != 0 ) {
        // INPUT/OUTPUT action
        ((IOAction)a).subject.offers.remove(a);
        }
      }
    }
  }
```

Clase `Lock`

```
class Lock {

  synchronized static boolean commit(Role p, Action pa, Role q, Action qa) {
    // Commits both roles or none
    if ( p.setCommitment(pa) ) {
      if ( q.setCommitment(qa) )
        return true;
      else
        p.resetCommitment();
      }
    return false;
    }
  }
```

Clase `State`

```
abstract class State {

  Role role;

  public State(Role role) {
    this.role = role;
    }

  public int select() {
    return role.select();
    }

  public Object perform() {
    return role.perform();
    }
```

```
  public Object perform(Action a) {
    role.select();
    return role.perform();
    }

  abstract public void attach(Link[] links);

  abstract public State spec();
  }
```

Clase `clase Zero`

```
class Zero extends State {

  public Zero(Role role) {
    super(role);
    }

  public void attach(Link[] links) {
    for(int i=0; i<links.length; i++)
      links[i].decr();
    }

  public State spec() {
    System.out.println(role.name + " stops");
    return null;
    }
  }
```

Clase `Link`

```
class Link {

    String name;
    Object datum;
    boolean written;
    ActionList offers;
    int nroles;
    LinkImp imp;

  public Link(String name) {
    this.name = name;
    this.datum = null;
    this.written = false;
    this.offers = new ActionList();
    int nroles = 0;
    imp = new ShrdMemLink(this);
    }
```

```
  public Link(String name, Object datum) {
    this(name);
    this.datum = datum;
    }

  public String toString() {
    return name;
    }

  public void incr() {
    nroles++;
    }

  public void decr() {
    nroles--;
    }

  public synchronized boolean write(Object datum) {
    return imp.write(datum);
    }

  public synchronized Object read() {
    return imp.read();
    }
  }
```

Clase `LinkImp`

```
abstract class LinkImp {

  Link theLink;

  public LinkImp(Link theLink) {
    this.theLink = theLink;
    }

  abstract public synchronized boolean write(Object datum);

  abstract public synchronized Object read();
  }
```

Clase `ShrdMemLink`

```
class ShrdMemLink extends LinkImp {

  public ShrdMemLink(Link theLink) {
    super(theLink);
    }
```

```
  public synchronized boolean write(Object datum) {
    if ( ! theLink.written ) {
      theLink.datum = datum;
      theLink.written = true;
      return true;
      }
    else {
      // Not read since written
      return false;
      }
    }

  public synchronized Object read() {
    if ( theLink.written ) {
      Object datum = theLink.datum;
      theLink.datum = null;
      theLink.written = false;
      return datum;
      }
    else {
      // Not written yet
      return null;
      }
    }
  }
```

Clase `Action`

```
abstract class Action {

    Role role;
    int number;
    int sign;

  public Action(Role role, int number, int sign) {
    this.role = role;
    this.number = number;
    this.sign = sign;
    role.offers.add(this); // Publishes the action in role's offers
    }

  abstract public String toString();

  abstract public Object perform();
  }
```

Clase `Tau`

```
class Tau extends Action {

  public Tau(Role role, int number) {
    super(role,number,0);
    }

  public String toString() {
    return "tau(" + number + ")";
    }

  public Object perform() {
    role.component.perform("tau" + number, null);
    return null;
    }
  }
```

Clase `IOAction`

```
abstract class IOAction extends Action {
    Link subject;
    Object object;

  public IOAction(Role role, int number, Link subject, int sign) {
    super(role, number, sign);
    this.subject = subject;
    subject.offers.add(this); // Publishes the action in link's offers
    }
  }
```

Clase `Input`

```
class Input extends IOAction {

  public Input(Role role, int number, Link subject) {
    super(role, number, subject, -1);
    this.object = new Link("");
    }

  public String toString() {
    return subject + "?(" + object + ")";
    }

  public Object perform() {
    while ( ( object  = subject.read() ) == null ) {
      role.t.yield();
      }
    role.component.perform(subject.name + "QUE", object);
    return object;
    }
  }
```

Clase `Output`

```
class Output extends IOAction {

  public Output(Role role, int number, Link subject, Object object) {
    super(role, number, subject, 1);
    this.object = object;
    }

  public Output(Role role, int number, Link subject) {
    this(role, number, subject, "");
    }

  public String toString() {
    return subject + "!(" + object + ")";
    }

  public Object perform() {
    object = role.component.perform(subject.name + "EXC", object);
    while ( ! subject.write(object) ) {
      role.t.yield();
      }
    while ( subject.written ) {
      role.t.yield(); // Waits for the reader to fulfil the action
      }
    return object;
    }
  }
```

Clase `MultiOutput`

```
class MultiOutput extends IOAction {

  public MultiOutput(Role role, int number, Link subject, Object object) {
    // The sign indicates the number of readers
    super(role, number, subject, subject.nroles-1);
    if ( sign == 0 ) {
      // To avoid confusion with TAU-actions
      // If there are no readers won't commit this action
      sign = 1;
      }
    this.object = object;
    }

  public MultiOutput(Role role, int number, Link subject) {
    this(role, number, subject, "");
    }
```

```
  public String toString() {
    return subject + "![*](" + object + ")";
    }

  public Object perform() {
    object = role.component.perform(subject.name + "BRD", object);
    if ( sign > 1 ) { // Real broadcast (more than one reader)
      role.offers.add(this); // Offers action again
      subject.offers.add(this);
      }
    Link ack = new Link(subject.name + "Ack");
    for(int i=1; i <= sign; i++) {
      Link l = new Link(subject.name + "[" + i + "]");
      while ( ! subject.write(l) ) {
        role.t.yield();
        }
      while ( subject.written ) {
        role.t.yield(); // Waits for the reader to fulfil the action
        }
      while ( ! l.write(object) ) {
        role.t.yield();
        }
      while ( l.written ) {
        role.t.yield(); // Waits for the reader to fulfil the action
        }
      while ( ! l.write(ack) ) {
        role.t.yield();
        }
      while ( l.written ) {
        role.t.yield(); // Waits for the reader to fulfil the action
        }
      // The role is released for a new commitment
      role.resetCommitment();
      }
    role.removeOffers();
    for(int i=1; i<= sign; i++) {
      Link acki = new Link("ack[" + i + "]");
      while ( ! ack.write(acki) )  {
        role.t.yield();
        }
      while ( ack.written ) {
        role.t.yield(); // Waits for the reader to fulfil the action
        }
      }
    return object;
    }
  }
```

Clase `MultiInput`

```
class MultiInput extends IOAction {

  public MultiInput(Role role, int number, Link subject) {
    super(role, number, subject, -1);
    this.object = new Link("");
    }

  public String toString() {
    return subject + "QUE(" + object + ")";
    }

  public Object perform() {
    Link l, ack;
    while ( ( l = (Link) subject.read() ) == null ) {
      role.t.yield();
      }
    while ( ( object = l.read() ) == null ) {
      role.t.yield();
      }
    while ( ( ack = (Link) l.read() ) == null ) {
      role.t.yield();
      }
    while ( ( ack.read() ) == null ) {
      role.t.yield();
      }
    role.component.perform(subject.name + "QUE", object);
    return object;
    }
  }
```

Clase `ActionList`

```
class ActionList {
    Action action;
    ActionList next;

  public ActionList() {
    action = null;
    next = null;
    }

  public synchronized void add(Action a) {
    ActionList newNode = new ActionList();
    newNode.action = this.action;
    newNode.next = this.next;
    this.action = a;
    this.next = newNode;
    }
```

```
public synchronized Action atRandom() {
  int size = 0;
  ActionList l = this;
  while ( l.action != null ) {
    size++;
    l = l.next;
    }
  if ( size > 0 ) {
    int n = (int)(1 + Math.random()*size );
    l = this;
    while ( n > 1 ) {
      l = l.next;
      n--;
      }
    return l.action;
    }
  else {
   // There are no actions in the list
   return null;
   }
  }

public synchronized void remove(Action a) {
  ActionList l = this;
  boolean found = false;
  while ( l != null && !found) {
    if ( l.action == a )
      found = true;
    else {
      l = l.next;
      }
    }
  if ( found ) {
    if ( l.next != null ) {
      l.action = l.next.action;
      l.next = l.next.next;
      }
    else {
      // The list is empty now
      l.action = null;
      l.next = null;
      }
    }
  }
```

```
public synchronized Action find(int sign) {
  Action action = null;
  ActionList l = this;
  while ( l.action != null && action == null ) {
    if ( l.action.sign * sign > 0 ) {
      // The action has the sign we're looking for
      action = l.action;
      }
    l = l.next;
    }
  return action;
  }
}
```

C.2 Sistema de Subastas

Clase `SistemaDeSubastas`

```
class SistemaDeSubastas extends Component {

  Component subastador;
  Component generador;
  Link conexion;
  Link precio, puja, acuse, vendido, item, desconexion;

  public SistemaDeSubastas(String name, int pid,
                           Component subastador,
                           Component generador) {
    super("SistemaDeSubastas " + name, pid);
    this.subastador = subastador;
    this.generador = generador;
    }

  public SistemaDeSubastas(String name, int pid) {
    this(name, pid, new Subastador("subastador", pid),
                                  new GeneradorDePostores("generador", pid));
    }

  public Object clone() {
    return new SistemaDeSubastas(name, pid, subastador, generador);
    }

  public void attach(Link[] links) {
    conexion = new Link("conexion");
    precio = new Link("precio");
    puja = new Link("puja");
    acuse = new Link("acuse");
    vendido = new Link("vendido");
    item = new Link("item");
    desconexion = new Link("desconexion");
    Link links1[] = {conexion, precio, puja, acuse, vendido, item, desconexion};
    subastador.attach(links1);
    Link links2[] = {conexion, precio, puja, acuse, vendido, item, desconexion};
    generador.attach(links2);
    }
  }
```

Clase Subastador

```
class Subastador extends Component {

  public int max;
  Role conectar;
  Role subastar;

  public Subastador(String name,int pid) {
    super("Subastador " + name, pid);
    max = 0;
    n = 0;
    conectar = new SubastadorConectar("conectar",pid,this);
    subastar = new Subastar("subastar",pid,this);
    }

  public Object clone() {
    return new Subastador(name, pid);
    }

  public void attach(Link links[]) {
    Link conexion = links[0];
    Link precio = links[1];
    Link puja = links[2];
    Link acuse = links[3];
    Link vendido = links[4];
    Link item = links[5];
    Link desconexion = links[6];
    Link links1[] = { conexion };
    conectar.attach(links1);
    Link links2[] = { precio, puja, acuse, vendido, item, desconexion };
    subastar.attach(links2);
    }
  }
```

Clase SubastadorConectar

```
class SubastadorConectar extends Role {
  public SubastadorConectar(String name, int pid, Component component) {
    super("SubastadorConectar " + name, pid, component);
    state = new SubastadorConectarInicial(this);
    }
  }
```

Clase SubastadorConectarInicial

```
class SubastadorConectarInicial extends State {

  Link conexion;

  public SubastadorConectarInicial(Role role) {
    super(role);
    }

  public void attach(Link links[]) {
    conexion = links[0];
    }

  public State spec() {
    perform(new Input(role,1,conexion));
    int max = ((Integer)role.component.get("max")).intValue();
    max++;
    role.component.set("max",new Integer(max));
    return this;
    }
  }
```

Clase Subastar

```
class Subastar extends Role {

  int valor;

  public Subastar(String name, int pid, Component component) {
    super("Subastar " + name, pid, component);
    state = new SubastarInicial(this);
    }
  }
```

Clase SubastarInicial

```
class SubastarInicial extends State {

  Link precio, puja, acuse, vendido, item, desc;

  public SubastarInicial(Role role) {
    super(role);
    }
```

```
  public void attach(Link links[]) {
    precio = links[0];
    puja = links[1];
    acuse = links[2];
    vendido = links[3];
    item = links[4];
    desc = links[5];
    }

  public State spec() {
    ((Subastar)role).valor = 100;
    perform(new MultiOutput(role, 0,precio,new Integer(((Subastar)role).valor)));
    Link links[] = { precio, puja, acuse, vendido, item, desc };
    State next = new SubastarSubastando(role);
    next.attach(links);
    return next;
    }
  }
```

Clase SubastarSubastando

```
class SubastarSubastando extends State {

  Link precio, puja, acuse, vendido, item, desc;

  public SubastarSubastando(Role role) {
    super(role);
    }

  public void attach(Link links[]) {
    precio = links[0];
    puja = links[1];
    acuse = links[2];
    vendido = links[3];
    item = links[4];
    desc = links[5];
    }

  public State spec() {
    new Tau(role,1);
    new Input(role,2,puja);
    new Input(role,3,acuse);
    new Input(role,4,desc);
    int choice = select();
    switch(choice) {
      case 1 :
        perform();
        perform(new MultiOutput(role, 0, precio,
                                new Integer(((Subastar)role).valor)));
        return this;
```

```
      case 2 :
        Link respuesta = (Link) perform();
        new Tau(role,21);
        new Tau(role,22);
        choice = select();
        switch(choice) {
          case 21 :
            perform();
            ((Subastar)role).valor+=10;
            perform(new Output(role,0,respuesta,"ACEPTAR"));
            new Tau(role,211);
            new Tau(role,212);
            choice = select();
            switch(choice) {
              case 211 :
                perform();
                perform(new MultiOutput(role,0,vendido));
                Link articulo = new Link("articulo");
                perform(new Output(role,0,item,articulo));
                Link links[] = { precio, puja, acuse, vendido, item, desc };
                State next = new SubastarInicial(role);
                next.attach(links);
                return next;
              case 212 :
                perform();
                perform(new MultiOutput(role, 0, precio,
                                        new Integer(((Subastar)role).valor)));
                return this;
              }
          case 22 :
            perform();
            perform(new Output(role,0,respuesta,"RECHAZAR"));
            return this;
          }
      case 3 :
        perform();
        return this;
      case 4 :
        perform();
        int max = ((Integer)role.component.get("max")).intValue();
        max--;
        role.component.set("max",new Integer(max));
        return this;
      }
    }
  }
```

Clase GeneradorDePostores

```
class GeneradorDePostores extends Component {

  Role conectar;
  public Vector postor;
  Component postorSeed;
  Link conexion;
  Link precio, puja, acuse, vendido, item, desc;

  public GeneradorDePostores(String name, int pid, Component postorSeed) {
    super("GeneradorDePostores " + name, pid);
    conectar = new GeneradorDePostoresConectar("conectar", pid, this);
    postor = new Vector();
    this.postorSeed = postorSeed;
    }

  public GeneradorDePostores(String name, int pid) {
    this(name, pid, new Postor("postor",pid));
    }

  public Object clone() {
    return new GeneradorDePostores(name, pid, postorSeed);
    }

  public void attach(Link links[]) {
    conexion = links[0];
    precio = links[1];
    puja = links[2];
    acuse = links[3];
    vendido = links[4];
    item = links[5];
    desc = links[6];
    Link links1[] = { conexion };
    conectar.attach(links1);
    }

  public void newPostor() {
    try {
      postorSeed.pid = postor.size() + 1;
      Component newComp = (Component) postorSeed.clone();
      postor.addElement(newComp);
      Link links[] = { precio, puja, acuse, vendido, item, desc};
      newComp.attach(links);
      }
    catch (Exception e) {
      System.out.println("Cannot create new postor");
      }
    }
  }
```

Clase GeneradorDePostoresConectar

```
class GeneradorDePostoresConectar extends Role {

  public GeneradorDePostoresConectar(String name, int pid, Component component) {
    super("GeneradorDePostores " + name, pid, component);
    state = new GeneradorDePostoresConectarInicial(this);
    }
  }
```

Clase GeneradorDePostoresConectarInicial

```
class GeneradorDePostoresConectarInicial extends State {

  Link conexion;

  public GeneradorDePostoresConectarInicial(Role role) {
    super(role);
    }

  public void attach(Link links[]) {
    conexion = links[0];
    }

  public State spec() {
    perform(new Output(role,1,conexion));
    ((GeneradorDePostores)role.component).newPostor();
    return this;
    }
  }
```

Clase Postor

```
class Postor extends Component {

  Role pujar;

  public Postor(String name, int pid) {
    super("Postor " + name, pid);
    pujar = new Pujar("pujar", pid, this);
    }

  public Object clone() {
    return new Postor(name, pid);
    }
```

```
  public void attach(Link links[]) {
    Link precio = links[0];
    Link puja = links[1];
    Link acuse = links[2];
    Link vendido = links[3];
    Link item = links[4];
    Link desc = links[5];
    Link links1[] = { precio, puja, acuse, vendido, item, desc };
    pujar.attach(links1);
    }
  }
```

Clase Pujar

```
class Pujar extends Role {

  Link id;

  public Pujar(String name, int pid, Component component) {
    super("Pujar " + name, pid, component);
    id = new Link("id",new Integer(pid));
    state = new PujarInicial(this);
    }
  }
```

Clase PujarInicial

```
class PujarInicial extends State {

  Link precio, puja, acuse, vendido, item, desc;

  public PujarInicial(Role role) {
    super(role);
    }

  public void attach(Link links[]) {
    precio = links[0];
    puja = links[1];
    acuse = links[2];
    vendido = links[3];
    item = links[4];
    desc = links[5];
    }
```

```
  public State spec() {
    new MultiInput(role,1,precio);
    new MultiInput(role,2,vendido);
    int choice = select();
    switch(choice) {
      case 1 : {
        Object valor = perform();
        Link links[] = { precio, puja, acuse, vendido, item, desc };
        State next = new PujarDecidiendo(role);
        next.attach(links);
        return next;
      case 2 :
        perform();
        return this;
      }
    }
  }
```

Clase `PujarDecidiendo`

```
class PujarDecidiendo extends State {

  Link precio, puja, acuse, vendido, item, desc;

  public PujarDecidiendo(Role role) {
    super(role);
    }

  public void attach(Link links[]) {
    precio = links[0];
    puja = links[1];
    acuse = links[2];
    vendido = links[3];
    item = links[4];
    desc = links[5];
    }

  public State spec() {
    new MultiInput(role,1,precio);
    new MultiInput(role,2,vendido);
    new Output(role,3,puja,((Pujar)role).id);
    new Output(role,4,desc,((Pujar)role).id);
    int choice = select();
    switch(choice) {
      case 1 :
        Object valor = perform();
        return this;
```

```
      case 2 :
        perform();
        Link links[] = { precio, puja, acuse, vendido, item, desc };
        State next = new PujarInicial(role);
        next.attach(links);
        return next;
      case 3 :
        perform();
        Link links[] = { precio, puja, acuse, vendido, item, desc };
        State next = new PujarEsperando(role);
        next.attach(links);
        return next;
      case 4 :
        perform();
        Link links[] = { precio, puja, acuse, vendido, item, desc };
        State next = new Zero(role);
        next.attach(links);
        return next;
      }
    }
  }
```

Clase `PujarEsperando`

```
class PujarEsperando extends State {

  Link precio, puja, acuse, vendido, item, desc;

  public PujarEsperando(Role role) {
    super(role);
    }

  public void attach(Link links[]) {
    precio = links[0];
    puja = links[1];
    acuse = links[2];
    vendido = links[3];
    item = links[4];
    desc = links[5];
    }

  public State spec() {
    String resultado = (String) perform(new Input(role,0,((Pujar)role).id));
    if ( resultado == "RECHAZAR" ) {
      Link links[] = { precio, puja, acuse, vendido, item, desc };
      State next = new PujarInicial(role);
      next.attach(links);
      return next;
      }
```

```
    if ( resultado == "ACEPTAR" )  {
      Link links[] = { precio, puja, acuse, vendido, item, desc };
      State next = new PujarGanando(role);
      next.attach(links);
      return next;
      }
    }
  }
```

Clase PujarGanando

```
class PujarGanando extends State {

  Link precio, puja, acuse, vendido, item, desc;

  public PujarGanando(Role role) {
    super(role);
    }

  public void attach(Link links[]) {
    precio = links[0];
    puja = links[1];
    acuse = links[2];
    vendido = links[3];
    item = links[4];
    desc = links[5];
    }

  public State spec() {
    new MultiInput(role,1,precio);
    new MultiInput(role,2,vendido);
    int choice = select();
    switch(choice) {
      case 1 :
        Object valor = perform();
        Link links[] = { precio, puja, acuse, vendido, item, desc };
        State next = new PujarDecidiendo(role);
        next.attach(links);
        return next;
      case 2 :
        perform();
        Object articulo = perform(new Input(role,0,item));
        Link links[] = { precio, puja, acuse, vendido, item, desc };
        State next = new PujarInicial(role);
        next.attach(links);
        return next;
      }
    }
  }
```

Clase `InstanceSubasta`

```
class InstanceSubasta {

  public static void main(String args[]) {
    new SistemaDeSubastas("sistema",0).attach(null);
    }
  }
```

C.3 Extensión del sistema

Clase `PostorLimitado`

```
class PostorLimitado extends Component {

  Role pujar;

  public PostorLimitado(String name, int pid) {
    super("PostorLimitado " + name, pid);
    pujar = new PujarLimitadamente("pujar", pid, this);
    }

  public Object clone() {
    return new PostorLimitado(name, pid);
    }

  public void attach(Link links[]) {
    Link precio = links[0];
    Link puja = links[1];
    Link acuse = links[2];
    Link vendido = links[3];
    Link item = links[4];
    Link desc = links[5];
    Link links1[] = { precio, puja, acuse, vendido, item, desc };
    pujar.attach(links1);
    }
  }
```

Clase `PujarLimitadamente`

```
class PujarLimitadamente extends Role {

  Link id;

  public PujarLimitadamente(String name, int pid, Component component) {
    super("PujarLimitadamente " + name, pid, component);
    id = new Link("id",new Integer(pid));
    state = new PujarLimitadamenteInicial(this);
    }
  }
```

Clase `PujarLimitadamenteInicial`

```
class PujarLimitadamenteInicial extends State {

  Link precio, puja, acuse, vendido, item, desc;

  public PujarLimitadamenteInicial(Role role) {
    super(role);
    }

  public void attach(Link links[]) {
    precio = links[0];
    puja = links[1];
    acuse = links[2];
    vendido = links[3];
    item = links[4];
    desc = links[5];
    }

  public State spec() {
    new MultiInput(role,1,precio);
    new MultiInput(role,2,vendido);
    int choice = select();
    switch(choice) {
      case 1 :
        Object valor = perform();
        Link links[] = { precio, puja, acuse, vendido, item, desc };
        State next = new PujarLimitadamenteDecidiendo(role);
        next.attach(links);
        return next;
      case 2 :
        perform();
        return this;
      }
    }
  }
```

Clase `PujarLimitadamenteDecidiendo`

```
class PujarLimitadamenteDecidiendo extends State {

  Link precio, puja, acuse, vendido, item, desc;

  public PujarLimitadamenteDecidiendo(Role role) {
    super(role);
    }
```

```
public void attach(Link links[]) {
  precio = links[0];
  puja = links[1];
  acuse = links[2];
  vendido = links[3];
  item = links[4];
  desc = links[5];
  }

public State spec() {
  new MultiInput(role,1,precio);
  new MultiInput(role,2,vendido);
  new Output(role,3,puja,((PujarLimitadamente)role).id);
  new Output(role,4,desc,((PujarLimitadamente)role).id);
  new Output(role,5,acuse);
  int choice = select();
  switch(choice) {
    case 1 :
      Object valor = perform();
      return this;
    case 2 :
      perform();
      Link links[] = { precio, puja, acuse, vendido, item, desc };
      State next = new PujarLimitadamenteInicial(role);
      next.attach(links);
      return next;
    case 3 :
      perform();
      Link links[] = { precio, puja, acuse, vendido, item, desc };
      State next = new PujarLimitadamenteEsperando(role);
      next.attach(links);
      return next;
    case 4 :
      perform();
      Link links[] = { precio, puja, acuse, vendido, item, desc };
      State next = new Zero(role);
      next.attach(links);
      return next;
    case 5 :
      perform();
      Link links[] = { precio, puja, acuse, vendido, item, desc };
      State next = new PujarLimitadamenteRehusando(role);
      next.attach(links);
      return next;
    }
  }
}
```

Clase PujarLimitadamenteEsperando

```
class PujarLimitadamenteEsperando extends State {

  Link precio, puja, acuse, vendido, item, desc;

  public PujarLimitadamenteEsperando(Role role) {
    super(role);
    }

  public void attach(Link links[]) {
    precio = links[0];
    puja = links[1];
    acuse = links[2];
    vendido = links[3];
    item = links[4];
    desc = links[5];
    }

  public State spec() {
    String resultado =
      (String) perform(new Input(role, 0, ((PujarLimitadamente)role).id));
    if ( resultado == "RECHAZAR" ) {
      Link links[] = { precio, puja, acuse, vendido, item, desc };
      State next = new PujarLimitadamenteInicial(role);
      next.attach(links);
      return next;
      }
    if ( resultado == "ACEPTAR" )  {
      Link links[] = { precio, puja, acuse, vendido, item, desc };
      State next = new PujarLimitadamenteGanando(role);
      next.attach(links);
      return next;
      }
    }
  }
```

Clase PujarLimitadamenteGanando

```
class PujarLimitadamenteGanando extends State {

  Link precio, puja, acuse, vendido, item, desc;

  public PujarLimitadamenteGanando(Role role) {
    super(role);
    }
```

```
  public void attach(Link links[]) {
    precio = links[0];
    puja = links[1];
    acuse = links[2];
    vendido = links[3];
    item = links[4];
    desc = links[5];
    }

  public State spec() {
    new MultiInput(role,1,precio);
    new MultiInput(role,2,vendido);
    int choice = select();
    switch(choice) {
      case 1 :
        Object valor = perform();
        Link links[] = { precio, puja, acuse, vendido, item, desc };
        State next = new PujarLimitadamenteDecidiendo(role);
        next.attach(links);
        return next;
      case 2 :
        perform();
        Object articulo = perform(new Input(role,0,item));
        Link links[] = { precio, puja, acuse, vendido, item, desc };
        State next = new PujarLimitadamenteInicial(role);
        next.attach(links);
        return next;
      }
    }
  }
```

Clase `LimitadamenteRehusando`

```
class PujarLimitadamenteRehusando extends State {

  Link precio, puja, acuse, vendido, item, desc;

  public PujarLimitadamenteRehusando(Role role) {
    super(role);
    }

  public void attach(Link links[]) {
    precio = links[0];
    puja = links[1];
    acuse = links[2];
    vendido = links[3];
    item = links[4];
    desc = links[5];
    }
```

```
  public State spec() {
    new MultiInput(role,1,precio);
    new MultiInput(role,2,vendido);
    int choice = select();
    switch(choice) {
      case 1 :
        Object valor = perform();
        new Output(role,11,acuse);
        new MultiInput(role,12,precio);
        new MultiInput(role,13,vendido);
        choice = select();
        switch(choice) {
          case 11 :
            perform();
            return this;
          case 12 :
            valor = perform();
            return this;
          case 13 :
            perform();
            Link links[] = { precio, puja, acuse, vendido, item, desc };
            State next = new PujarLimitadamenteInicial(role);
            next.attach(links);
            return next;
          }
      case 2 :
        perform();
        Link links[] = { precio, puja, acuse, vendido, item, desc };
        State next = new PujarLimitadamenteInicial(role);
        next.attach(links);
        return next;
      }
    }
  }
```

Clase `SubastadorJusto`

```
class SubastadorJusto extends Component {

  public int max;
  Role conectar;
  Role subastar;

  public SubastadorJusto(String name,int pid) {
    super("SubastadorJusto " + name, pid);
    max = 0;
    conectar = new SubastadorJustoConectar("conectar",pid,this);
    subastar = new SubastarJustamente("subastar",pid,this);
    }
```

```
  public Object clone() {
    return new SubastadorJusto(name, pid);
    }

  public void attach(Link links[]) {
    Link conexion = links[0];
    Link precio = links[1];
    Link puja = links[2];
    Link acuse = links[3];
    Link vendido = links[4];
    Link item = links[5];
    Link desconexion = links[6];
    Link links1[] = { conexion };
    conectar.attach(links1);
    Link links2[] = { precio, puja, acuse, vendido, item, desconexion };
    subastar.attach(links2);
    }
  }
```

Clase SubastadorJustoConectar

```
class SubastadorJustoConectar extends Role {

  public SubastadorJustoConectar(String name, int pid, Component component) {
    super("SubastadorJustoConectar " + name, pid, component);
    state = new SubastadorJustoConectarInicial(this);
    }
  }
```

Clase SubastadorJustoConectarInicial

```
class SubastadorJustoConectarInicial extends State {

  Link conexion;

  public SubastadorJustoConectarInicial(Role role) {
    super(role);
    }

  public void attach(Link links[]) {
    conexion = links[0];
    }

  public State spec() {
    perform(new Input(role,1,conexion));
    int max = ((Integer)role.component.get("max")).intValue();
    max++;
    role.component.set("max",new Integer(max));
    return this;
    }
  }
```

Clase SubastarJustamente

```
class SubastarJustamente extends Role {

  int valor;
  int n;

  public SubastarJustamente(String name, int pid, Component component) {
    super("SubastarJustamente " + name, pid, component);
    state = new SubastarJustamenteInicial(this);
    int max = ((Integer)component.get("max")).intValue();
    n = max;
    }
  }
```

Clase SubastarJustamenteInicial

```
class SubastarJustamenteInicial extends State {

  Link precio, puja, acuse, vendido, item, desc;

  public SubastarJustamenteInicial(Role role) {
    super(role);
    }

  public void attach(Link links[]) {
    precio = links[0];
    puja = links[1];
    acuse = links[2];
    vendido = links[3];
    item = links[4];
    desc = links[5];
    }

  public State spec() {
    ((SubastarJustamente)role).valor = 100;
    perform(new MultiOutput(role, 0, precio,
                            new Integer(((SubastarJustamente)role).valor)));
    Link links[] = { precio, puja, acuse, vendido, item, desc };
    State next = new SubastarJustamenteSubastando(role);
    next.attach(links);
    return next;
    }
  }
```

Clase SubastarJustamenteSubastando

```
class SubastarJustamenteSubastando extends State {

  Link precio, puja, acuse, vendido, item, desc;

  public SubastarJustamenteSubastando(Role role) {
    super(role);
    }

  public void attach(Link links[]) {
    precio = links[0];
    puja = links[1];
    acuse = links[2];
    vendido = links[3];
    item = links[4];
    desc = links[5];
    }

  public State spec() {
    new Tau(role,1);
    new Input(role,2,puja);
    new Input(role,3,acuse);
    new Input(role,4,desc);
    int choice = select();
    switch(choice) {
      case 1 :
        perform();
        perform(new MultiOutput(role, 0, precio,
                                new Integer(((SubastarJustamente)role).valor)));
        int max = ((Integer)role.component.get("max")).intValue();
        ((SubastarJustamente)role).n = 0;
        return this;
      case 2 :
        Link respuesta = (Link) perform();
        ((SubastarJustamente)role).n++;
        new Tau(role,21);
        new Tau(role,22);
        choice = select();
        switch(choice) {
          case 21 :
            perform();
            ((SubastarJustamente)role).valor+=10;
            perform(new Output(role,0,respuesta,"ACEPTAR"));
            int max = ((Integer)role.component.get("max")).intValue();
            if (((SubastarJustamente)role).n ==  max) {
              perform(new MultiOutput(role,0,vendido));
              Link articulo = new Link("articulo");
              perform(new Output(role,0,item,articulo));
```

```
            ((SubastarJustamente)role).n = 0;
            Link links[] = { precio, puja, acuse, vendido, item, desc };
            State next = new SubastarJustamenteInicial(role);
            next.attach(links);
            return next;
            }
          max = ((Integer)role.component.get("max")).intValue();
          if (((SubastarJustamente)role).n < max) {
            perform(new MultiOutput(role, 0, precio,
                            new Integer(((SubastarJustamente)role).valor)));
            ((SubastarJustamente)role).n = 0;
            return this;
            }
        case 22 :
          perform();
          perform(new Output(role,0,respuesta,"RECHAZAR"));
          return this;
      }
    case 3 :
      perform();
      ((SubastarJustamente)role).n++;
      return this;
    case 4 :
      perform();
      int max = ((Integer)role.component.get("max")).intValue();
      max--;
      role.component.set("max",new Integer(max));
      return this;
    }
  return new Zero(role);
  }
}
```

Clase `InstanceSubastaJustaLimitada`

```
class InstanceSubastaJustaLimitada {

  public static void main(String args[]) {
    new SistemaDeSubastas("sistema", 0,
                    new SubastadorJusto("subastador", 0),
                    new GeneradorDePostores("generador", 0,
                                            new PostorLimitado("postor", 0))
                    ).attach(null);
  }
}
```

Bibliografía

[Abowd et al., 1993] Abowd, G., Allen, R., y Garlan, D. (1993). Using Style to Understand Descriptions of Software Architecture. En *ACM Foundations of Software Engineering (FSE'93)*.

[Aceto y Hennessy, 1992] Aceto, L. y Hennessy, M. (1992). Termination, Deadlock, and Divergence. *Journal of the ACM*, 39(1):147–187.

[Aceto y Hennessy, 1994] Aceto, L. y Hennessy, M. (1994). Adding Action Refinement to a Finite Process Algebra. *Information and Computation*, 115(2):179–247.

[Aksit et al., 1993] Aksit, M. et al. (1993). Abstracting Object Interactions Using Composition Filters. En *7th European Conf. on Object-Oriented Programming (ECOOP'93)*, núm. 791 de LNCS, pp. 152–184. Springer Verlag.

[Allen et al., 1998] Allen, R., Doucence, R., y Garlan, D. (1998). Specifying and Analyzing Dynamic Software Architectures. En *ETAPS'98*, Lisboa (Portugal).

[Allen y Garlan, 1992] Allen, R. y Garlan, D. (1992). A Formal Approach to Software Architectures. En *IFIP Congress'92*.

[Allen y Garlan, 1994] Allen, R. y Garlan, D. (1994). Formalizing architectural connection. En *16th Int. Conf. on Software Engineering (ICSE'94)*, pp. 71–80, Sorrento (Italia).

[Allen y Garlan, 1997] Allen, R. y Garlan, D. (1997). A Formal Basis for Architectural Connection. *ACM Trans. on Software Engineering and Methodology*, 6(3):213–49.

[America, 1991] America, P. (1991). Designing an Object-Oriented Programming Language with Behavioral Subtyping. En *Foundations of Object-Oriented Languages*, núm. 489 de LNCS, pp. 60–90. Springer Verlag.

[Back y von Wright, 1998] Back, R. J. R. y von Wright, J. (1998). *Refinement Calculus. A Systematic Introduction*. Springer Verlag.

[Baeten, 1990] Baeten, J. C. M. (ed.) (1990). *Applications of Process Algebra*. Cambridge University Press.

[Baeten y Bergstra, 1991] Baeten, J. C. M. y Bergstra, J. A. (1991). Real time process algebra. *Formal aspects of computing*, 3(2):142–188.

[Bass et al., 1998] Bass, L., Clements, P., y Kazman, R. (1998). *Software Architecture in Practice*. Addison Wesley.

[Bastide et al., 1999] Bastide, R., Sy, O., y Palanque, P. (1999). Formal Specification and Prototyping of CORBA Systems. En *13th European Conf. on Object-Oriented Programming (ECOOP'99)*, núm. 1628 de LNCS, pp. 474–494. Springer Verlag.

[Beach, 1992] Beach, B. W. (1992). Connecting Software Components with Declarative Glue. En *14th Int. Conf. on Software Engineering (ICSE'92)*.

[Bergstra y Klint, 1998] Bergstra, J. A. y Klint, P. (1998). The Discrete Time TOOLBUS — A Software Coordination Architecture. *Science of Computer Programming*, 31(2–3):179–203.

[Bergstra y Klop, 1985] Bergstra, J. A. y Klop, J. W. (1985). Algebra of Communicating Processes with Abstraction. *Theoretical Computer Science*, 37(1):77–121.

[Berry y Boudol, 1992] Berry, G. y Boudol, G. (1992). The Chemical Abstract Machine. *Theoretical Computer Science*, 96(1):217–248.

[Boehm, 1988] Boehm, B. (1988). A Spiral Model for Software Development and Enhancement. *Computer*, 21(5):61–72.

[Booch et al., 1999] Booch, G. et al. (1999). *The Unified Modeling Language User Guide*. Addison Wesley.

[Bosch, 1997] Bosch, J. (1997). Design Patterns & Frameworks: On the Issue of Language Support. En *ECOOP'97 Workshop Reader*, núm. 1357 de LNCS. Springer Verlag.

[Bosch, 1998] Bosch, J. (1998). Design Patterns as Language Constructs. *Journal of Object-Oriented Programming*, pp. 18–32.

[Bosch, 2000] Bosch, J. (2000). *Design & Use of Software Architectures: adapting and evolving a product-line approach*. Addison Wesley.

[Boudol, 1987] Boudol, G. (1987). Notes on Algebraic Calculi of Processes. En *Logics and Models of Concurrent Systems*, núm. 13 de NATO ASI series. K.Apt.

[Boudol, 1992] Boudol, G. (1992). Asynchrony and the π-calculus. Informe Técnico 1702, INRIA, Sofia Antipolis (Francia).

[Boudol, 1994] Boudol, G. (1994). Some Chemical Abstract Machines. En *A Decade of Concurrency*, núm. 803 de LNCS, pp. 92–123. Springer Verlag.

[Box, 1998] Box, D. (1998). *Essential COM*. Addison Wesley.

[Bracciali et al., 2001] Bracciali, A., Brogi, A., y Turini, F. (2001). Coordinating Interaction Patterns. En *ACM Symposium on Applied Computing (SAC'2001)*. ACM Press. (Aceptado).

[Breu et al., 1997] Breu, R. et al. (1997). Towards a Formalization of the Unified Modeling Language. En *11th European Conf. on Object-Oriented Programming (ECOOP'97)*, núm. 1241 de LNCS. Springer Verlag.

[Brogi y Ciancarini, 1991] Brogi, A. y Ciancarini, P. (1991). The concurrent language Shared Prolog. *ACM Trans. on Programming Languages and Systems*, 13(1):99–123.

[Brogi y Jacquet, 1999] Brogi, A. y Jacquet, J. M. (1999). On the Expressiveness of Coordination Models. En *3rd Int. Conf. on Coordination Languages and Models*, núm. 1594 de LNCS, pp. 134–149. Springer Verlag.

[Brown y Wallnau, 1998] Brown, A. W. y Wallnau, K. C. (1998). The Current State of CBSE. *IEEE Software*, 15(5):37–47.

[Broy et al., 1998] Broy, M. et al. (1998). What characterizes a (software) component? *Software Concepts and Tools*, 19:49–56.

[Bruns, 1997] Bruns, G. (1997). *Distributed Systems Analysis with CCS*. Prentice Hall.

[Buechi y Weck, 1999] Buechi, M. y Weck, W. (1999). The Greybox Approach: When Blackbox Specifications Hide Too Much. Informe Técnico TUCS-TR-297a, Turku, (Finlandia).

[Canal et al., 1996] Canal, C., Pimentel, E., y Troya, J. M. (1996). Software Architecture Specification with π-calculus. En *I Jornadas de Ingeniería del Software (JIS'96)*, pp. 31–40, Sevilla (España).

[Canal et al., 1997a] Canal, C., Pimentel, E., y Troya, J. M. (1997a). A Formal Specification Language for the Description of Architectural Patterns. En *ASOO'97*, pp. 47–56, Buenos Aires (Argentina).

[Canal et al., 1997b] Canal, C., Pimentel, E., y Troya, J. M. (1997b). LEDA: A Specification Language for Software Architecture. En *II Jornadas de Ingeniería del Software (JIS'97)*, pp. 207–219, San Sebastián (España).

[Canal et al., 1997c] Canal, C., Pimentel, E., y Troya, J. M. (1997c). On the Composition and Extension of Software Systems. En *FSE'97 Workshop on Foundations of Component-Based Systems*, pp. 50–59, Zurich (Suiza).

[Canal et al., 1998] Canal, C., Fuentes, L., Pimentel, E., y Troya, J. M. (1998). Component-Oriented Development of Interactive Systems. En *I Jornadas Iberoamericanas de Ingenieria de Requisitos y Ambientes Software (IDEAS'98)*, pp. 14–25, Torres (Brasil).

[Canal et al., 1999a] Canal, C., Fuentes, L., Pimentel, E., y Troya, J. M. (1999a). Coordinación de componentes distribuidos: un enfoque generativo basado en Arquitectura del Software. En *IV Jornadas de Ingeniería del Software (JISBD'99)*, pp. 443–454, Cáceres (España).

[Canal et al., 1999b] Canal, C., Fuentes, L., Troya, J. M., y Vallecillo, A. (1999b). Adding Semantic Information to IDLs. Is it Really Practical? En *OOPSLA'99 Workshop on Behavioral Semantics*, pp. 22–31, Denver (Estados Unidos).

[Canal et al., 1999c] Canal, C., Pimentel, E., y Troya, J. M. (1999c). Conformance and Refinement of Behavior in π-calculus. En *2nd Int. Workshop on Components-Based Software Development in Computational Logic*, pp. 1–12, Paris (Francia).

[Canal et al., 1999d] Canal, C., Pimentel, E., y Troya, J. M. (1999d). Specification and Refinement of Dynamic Software Architectures. En *Software Architecture*, pp. 107–126. Kluwer Academic Publishers.

[Canal et al., 1999e] Canal, C., Pimentel, E., y Troya, J. M. (1999e). Specification of Interacting Software Components: a Case Study. En *II Jornadas Iberoamericanas de Ingenieria de Requisitos y Ambientes Software (IDEAS'99)*, pp. 381–392, San José (Costa Rica).

[Canal et al., 2000] Canal, C., Fuentes, L., Troya, J. M., y Vallecillo, A. (2000). Extending CORBA Interfaces with π-calculus for Protocol Compatibility. En *TOOLS Europe 2000*, pp. 208–225, Mont Saint–Michel (Francia). IEEE Computer Society.

[Canal et al., 2001] Canal, C., Pimentel, E., y Troya, J. M. (2001). Compatibility and Inheritance in Software Architectures. *Science of Computer Programming*. (En imprenta).

[Cardelli y Gordon, 2000] Cardelli, L. y Gordon, A. D. (2000). Mobile Ambients. *Theoretical Computer Science*, 240(1):177–213.

[CCITT/ITU-T, 1988] CCITT/ITU-T (1988). Redes de Comunicación de Datos. Interconexión de Sistemas Abiertos: Modelo y Notación. Libro Azul. Recomendaciones de la serie X.200.

[Chandy et al., 1983] Chandy, K. M. et al. (1983). Distributed Deadlock Detection. *ACM Trans. on Computer Systems*, 1:144–156.

[Chapman et al., 1995] Chapman, M. et al. (1995). Software Architecture for the Future Information Market. *Computer Communications*, 18(11):825–837.

[Ciancarini et al., 1995] Ciancarini, P., Jensen, K., y Yankelewich, D. (1995). On the Operational Semantics of a Coordination Language. En Ciancarini, P., Nierstrasz, O., y Yonezawa, A. (eds.), *Object-Based Models and Languages for Concurrent Systems*, volumen 924 de *LNCS*, pp. 77–106. Springer Verlag.

[Ciancarini et al., 1998] Ciancarini, P., Mazza, M., y Pazzaglia, L. (1998). A Logic for a Coordination Model with Multiple Spaces. *Science of Computer Programming*, 31(2-3):205–229.

[Clark y Gregory, 1986] Clark, K. y Gregory, S. (1986). PARLOG: parallel programming in logic. *ACM Trans. on Programming Languages and Systems*, 8(1):1–49.

[Clavel et al., 1999] Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., y Quesada, J. (1999). Maude: Specification and Programming in Rewriting Logic. SRI International. http://maude.csl.sri.com.

[Clavel et al., 2000] Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., y Quesada, J. (2000). Maude: Specification and Programming in Rewriting Logic. *Theoretical Computer Science*. En imprenta.

[Codenie et al., 1997] Codenie, W. et al. (1997). From Custom Applications to Domain-Specific Frameworks. *Communications of the ACM*, 40(10):71–77.

[Collins-Cope, 2000] Collins-Cope, M. (2000). What is Software Architecture? Notes from a Panel Discussion at TOOLS Europe 2000.

[Compare et al., 1999] Compare, D., Inverardi, P., y Wolf, A. L. (1999). Uncovering Architectural Mismatch in Component Behavior. *Science of Computer Programming*, 33(2):101–131.

[Consortium, 1995] Consortium, T. (1995). Overall Concepts and Principles of TINA.

[Cook et al., 1990] Cook, W. R., Hill, W., y Canning, P. S. (1990). Inheritance is not subtyping. En *17th ACM symposium on Principles of programming languages (POPL'90)*, pp. 125–135, San Francisco (Estados Unidos). ACM Press.

[Davies y Schneider, 1992] Davies, J. y Schneider, S. (1992). A Brief History of Timed CSP. Informe Técnico PRG-96, Programming Research Group, Oxford University.

[de Boer y Palamidessi, 1992] de Boer, F. y Palamidessi, C. (1992). A Process Algebra for Concurrent Constraint Programming. En *Joint Int. Conf. and Symposium on Logic Programming*, pp. 463–477. MIT Press.

[Dhara y Leavens, 1996] Dhara, K. K. y Leavens, G. T. (1996). Forcing Behavioural Subtyping through specification inheritance. En *18th Int. Conf. on Software Engineering (ICSE'96)*, pp. 258–267, Berlin (Alemania). IEEE Press.

[Ducasse y Richner, 1997] Ducasse, S. y Richner, T. (1997). Executable Connectors: Towards Reusable Design Elements. En *ACM Foundations of Software Engineering (ESEC/FSE'97)*, núm. 1301 de LNCS. Springer Verlag.

[Duke et al., 1994] Duke, R., Rose, G., y Smith, G. (1994). Object-Z: A Specification Language for the Description of Standards. Informe Técnico 94–45, University of Queensland (Australia).

[Engberg y Nielsen, 1986] Engberg, U. y Nielsen, M. (1986). A Calculus of Communicating Systems with Label-passing. Informe Técnico DAIMI PB-208, Computer Science Dept., University of Århus, (Dinamarca).

[Evans y Kent, 1999] Evans, A. S. y Kent, S. (1999). Meta-modelling semantics of UML: the pUML approach. En *2nd Int. Conf. on the Unified Modeling Language*, núm. 1999 de LNCS. Springer Verlag.

[Fayad y Schmidt, 1997] Fayad, M. y Schmidt, D. (1997). Object-Oriented Application Frameworks. *Communications of the ACM*, 40(10):32–38.

[Fayad et al., 1999] Fayad, M., Schmidt, D., y Johnson, R. E. (1999). *Application Frameworks*. Wiley & Sons. (3 vol.).

[Feijs, 1999] Feijs, L. M. G. (1999). Modelling Microsof COM using π-calculus. En *Formal Methods'99*, núm. 1709 de LNCS, pp. 1343–1363. Springer Verlag.

[Formal Systems, 1992] Formal Systems (1992). Failures Divergence Refinement: User manual and tutorial. Informe técnico, Formal Systems (Europe) Ltd., Oxford (Reino Unido).

[Franch, 1997] Franch, X. (1997). The Convenience for a Notation to Express Non-Functional Characteristics of Software Components. En *FSE'97 Workshop on Foundations of Component-Based Systems*, pp. 101–110, Zurich (Suiza).

[Franch y Botella, 1998] Franch, X. y Botella, P. (1998). Putting Non-Functional Requirements into Software Architecture. En *9th IEEE Int. Workshop on Software Specification and Design (IWSSD)*, pp. 60–67, Ise-shima (Japón).

[Fuentes, 1998] Fuentes, L. (1998). *Una arquitectura de software para el desarrollo de servicios avanzados de telecomunicación*. Tesis Doctoral, Universidad de Málaga, Departamento de Lenguajes y Ciencias de la Computación.

[Fuentes y Troya, 2001] Fuentes, L. y Troya, J. M. (2001). Coordinating Distributed Components on the Web: an Integrated Development Environment. *Software Practice & Experience*. (En imprenta).

[Gamma et al., 1995] Gamma, E. et al. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley.

[Garlan, 1995] Garlan, D. (1995). Research Directions in Software Architecture. *ACM Computing Surveys*, 27(2):257–261.

[Garlan et al., 1997] Garlan, D., Monroe, R. T., y Wile, D. (1997). Acme: An Architecture Description Interchange Language. En *CASCON'97*, pp. 169–183, Toronto (Canadá).

[Garlan y Perry, 1995] Garlan, D. y Perry, D. E. (1995). Special Issue on Software Architecture. *IEEE Transqctions on Software Engineering*, 21(4).

[Gaspari y Zabattaro, 1999] Gaspari, M. y Zabattaro, G. (1999). An algebraic specification of the new asynchronous CORBA Messaging service. En *13th European Conf. on Object-Oriented Programming (ECOOP'99)*, núm. 1628 de LNCS, pp. 495–518. Springer Verlag.

[Gelernter, 1985] Gelernter, D. (1985). Generative Communication in Linda. *ACM Trans. on Programming Languajes and Systems*, 7(1):80–112.

[Gelernter y Carriero, 1992] Gelernter, D. y Carriero, N. (1992). Coordination languages and their significance. *Communications of the ACM*, 33(2):97–107.

[Giannakopoulou et al., 1999] Giannakopoulou, D., Kramer, J., y Cheung, S. C. (1999). Behaviour Analysis of Distributed Systems using the Tracta Approach. *Journal of Automated Software Engineering*, 6(1):7–35.

[GISUM, 1995] GISUM (1995). EVP: Un entorno para la integración de técnicas de descripción formal. En *I Jornadas de Informática*, pp. 607–616, Tenerife (España).

[Goguen et al., 2000] Goguen, J., Winkler, T., Meseguer, J., Futatsugi, K., y Jouannaud, J.-P. (2000). Introducing OBJ. En Goguen, J. y Malcolm, G. (eds.), *Software Engineering with OBJ: Algebraic Specification in Action*. Kluwer.

[Gorrieri y Rensik, 1997] Gorrieri, R. y Rensik, A. (1997). Action Refinement as an Implementation Relation. En *FASE'97*, núm. 1214 de LNCS, pp. 772–786. Springer Verlag.

[Gorrieri y Rensink, 2000] Gorrieri, R. y Rensink, A. (2000). Action Refinement. En Bergstra, J., Ponse, A., y Smolka, S. (eds.), *Handbook of Process Algebra*. Elsevier.

[Hartel et al., 1997] Hartel, P. et al. (1997). Information Systems Modelling with TROLL: Formal Methods at Work. *Information Systems*, 22(2/3):79–99.

[Hatley y Pirbhai, 1987] Hatley, D. y Pirbhai, I. A. (1987). *Strategies for Real-Time Systems Specification*. Dorset House.

[Henderson, 1997] Henderson, P. (1997). Formal Models of Process Components. En *FSE'97 Workshop on Foundations of Component-Based Systems*, pp. 131–140, Zurich (Suiza).

[Henderson, 1998] Henderson, P. (1998). From Formal Models to Validated Components in an Evolving System. Informe técnico, University of Southampton (Reino Unido).

[Henderson-Sellers et al., 1999] Henderson-Sellers, B. et al. (1999). Are Components Objects? En *OOPSLA'99 Panel Discussions*.

[Hirsch et al., 1999] Hirsch, D., Inverardi, P., y Montanari, U. (1999). Modeling Software Architectures and Styles with Graph Grammars and Constraint Solving. En *Software Architecture*, pp. 127–143. Kluwer Academic Publishers.

[Hoare, 1985] Hoare, C. A. R. (1985). *Communicating Sequential Processes*. Prentice Hall.

[Hofmeister et al., 1999] Hofmeister, C., Nord, R., y Soni, D. (1999). Describing Software Architectures with UML. En *Software Architecture*, pp. 145–159. Kluwer Academic Publishers.

[Honda y Tokoro, 1991] Honda, K. y Tokoro, M. (1991). An Object Calculus for Asynchronous Communication. En *5th European Conf. on Object-Oriented Programming (ECOOP'91)*, núm. 512 de LNCS, pp. 133–147. Springer Verlag.

[Honda et al., 1998] Honda, K., Vasconcelos, V. T., y Kubo, M. (1998). Language primitives and type disciplines for structured communication-based programming. En *European Symposium on Programming (ESOP'98)*, volumen 1381 de *LNCS*, pp. 122–138. Springer Verlag.

[Horita y Mano, 1996] Horita, E. y Mano, K. (1996). Nepi: A Network Programming Language Based on the π-Calculus. En *Coordination Languages and Models*, núm. 1061 de LNCS, pp. 424–427. Springer Verlag.

[Inverardi y Wolf, 1995] Inverardi, P. y Wolf, A. L. (1995). Formal Specification and Analysis of Software Architectures Using the Chemical Abstract Machine Model. *IEEE Trans. on Software Engineering*, 21(4):373–386.

[ISO, 1989] ISO (1989). Information Processing Systems – Open Systems Interconnection. LOTOS, a Formal Desciption Technique Based on the Temporal Ordering of Observational Behavior. Rec. ISO 8807, International Organization for Standarization.

[ISO/ITU-T, 1995] ISO/ITU-T (1995). Reference Model for Open Distributed Processing. Rec. ISO/IEC 10746-1/4, ITU-T X.901/4, International Organization for Standarization.

[Jacobson et al., 1999] Jacobson, I. et al. (1999). *The Unified Software Development Process*. Addison Wesley.

[Johnson, 1997] Johnson, E. R. (1997). Frameworks = Components + Patterns. *Communications of the ACM*, 40(10):39–42.

[Jones, 1993] Jones, C. B. (1993). A π-calculus Semantics of an Object-Based Design Notation. En *CONCUR'93*, núm. 715 de LNCS, pp. 158–172. Springer Verlag.

[Josephs, 1992] Josephs, M. (1992). Receptive Process Theory. *Acta Informatica*, 29(2):17–32.

[Josephs et al., 1989] Josephs, M. B., Hoare, C. A. R., y Jifeng, H. (1989). A Theory of Asynchronous Processes. Informe Técnico TR-6-89, Programming Research Group, Oxford University (Reino Unido).

[Knapp, 1987] Knapp, E. (1987). Deadlock Detection in Distributed Databases. *ACM Computing Surveys*, 19:303–328.

[Kramer, 1990] Kramer, J. (1990). Configuration Programming —A Framework for the Development of Distributable Systems. En *IEEE Conf. on Computer Systems and Software Engineering*, Israel. IEEE Press.

[Krieger y Adler, 1998] Krieger, D. y Adler, R. M. (1998). The Emergence of Distributed Component Platforms. *Computer Journal*, 41(3):43–53.

[Kruchten, 1995] Kruchten, P. B. (1995). The 4+1 View Model of Architecture. *IEEE Software*.

[Lamport, 1991] Lamport, L. (1991). The Temporal Logic of Actions. Informe Técnico 79, DEC Corporation, Systems Research Centre.

[Lamport, 1994] Lamport, L. (1994). TLZ. En *Workshops in Computing*, pp. 267–268. Springer Verlag.

[Lange y Nakamura, 1995] Lange, D. B. y Nakamura, Y. (1995). Interactive Visualization of Design Patterns Can Help in Framework Understanding. En *OOPSLA'95*, pp. 342–357.

[Larsen, 1987] Larsen, K. G. (1987). A context dependent equivalence between processes. *Theoretical Computer Science*, 49(2–3):185–215.

[Lea, 1995] Lea, D. (1995). Interface-Based Protocol Specification of Open Systems using PSL. En *9th European Conf. on Object-Oriented Programming (ECOOP'95)*, núm. 1241 de LNCS. Springer Verlag.

[Leavens y Dhara, 2000] Leavens, G. T. y Dhara, K. K. (2000). Concepts of Behavioral Subtyping and a Sketch of Their Extension to Component-Based Systems. En *Foundations of Component-Based Systens*, pp. 113–136. Cambridge University Press.

[LeMétayer, 1998] LeMétayer, D. (1998). Describing Software Architecture Styles using Graph Grammars. *IEEE Trans. on Software Engineering*, 24(7).

[Liskov y Wing, 1994] Liskov, B. y Wing, J. M. (1994). A behavioral notion of subtyping. *ACM Trans. on Programming Languages and Systems*, 16(6):1811–1841.

[Luckham et al., 1995] Luckham, D. C. et al. (1995). Specification and Analysis of System Architecture using Rapide. *IEEE Trans. on Software Engineering*, 21(4):336–355.

[Luckham et al., 2000] Luckham, D. C. et al. (2000). Key Concepts in Arquitecture Definition Languages. En Leavens, G. T. y Sitaraman, M. (eds.), *Foundations of Component-Based Systems*, pp. 23–46. Cambridge University Press.

[Lumpe, 1999] Lumpe, M. (1999). *A Pi-Calculus Based Approach to Software Composition.* Tesis Doctoral, University of Bern, Institute of Computer Science and Applied Mathematics.

[Magee et al., 1995] Magee, J., Eisenbach, S., y Kramer, J. (1995). Modeling Darwin in the π-calculus. En *Theory and Practice in Distributed Systems*, núm. 938 de LNCS, pp. 133–152. Springer Verlag.

[Magee y Kramer, 1996] Magee, J. y Kramer, J. (1996). Dynamic Structure in Software Architectures. En *ACM Foundations of Software Engineering (FSE'96)*, pp. 3–14, San Francisco (Estados Unidos).

[Magee et al., 1999] Magee, J., Kramer, J., y Giannakopoulou, D. (1999). Behaviour Analysis of Software Architectures. En *Software Architecture*, pp. 35–49. Kluwer Academic Publishers.

[Magee et al., 1989] Magee, J., Kramer, J., y Sloman, M. (1989). Constructing Distributed Systems in Conic. *IEEE Trans. on Software Engineering*, 15(6).

[Mak, 1992] Mak, V. W. (1992). Connection: An inter-component communication paradigm for configurable distributed systems. En *Workshop on Configurable Distributed Systems*, Londres (Reino Unido).

[May, 1985] May, D. (1985). Occam. *SIGPLAN Notices*, 18(4):69–79.

[Medvidovic et al., 1996] Medvidovic, N. et al. (1996). Using Object-Oriented Typing to Support Architectural Design in the C2 Style. En *ACM Foundations of Software Engineering (FSE'96)*, pp. 24–32, San Francisco (Estados Unidos).

[Medvidovic y Rosenblum, 1997] Medvidovic, N. y Rosenblum, D. S. (1997). Domains of Concern in Software Architectures and Architecture Description Languages. En *USENIX Conf. on Domain-Specific Languages*, Santa Barbara (Estados Unidos).

[Medvidovic y Rosenblum, 1999] Medvidovic, N. y Rosenblum, D. S. (1999). Assessing the Suitability of a Standard Design Method for Modeling Software Architectures. En *Software Architecture*, pp. 171–182. Kluwer Academic Publishers.

[Medvidovic et al., 1998] Medvidovic, N., Rosenblum, D. S., y Taylor, R. (1998). A Type Theory for Software Architectures. Informe Técnico UCI-ICS-98-14, Dept. of Information and Computer Science, University of California, Irvine (Estados Unidos).

[Medvidovic y Taylor, 2000] Medvidovic, N. y Taylor, R. (2000). A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Trans. on Software Engineering*, 26(1):70–93.

[Merino y Troya, 1996] Merino, P. y Troya, J. M. (1996). EVP: Integration of FDTs for the Analysis and Verification of Communication Protocols. En *Computer-Aided Verification CAV'96*, núm. 1102 de LNCS, pp. 406–410. Springer Verlag.

[Meyer, 1999] Meyer, B. (1999). On to components. *IEEE Computer*, 32(1).

[Microsoft, 1996] Microsoft (1996). DCOM - The Distributed Component Object Model.

[Mikhajlov y Sekerinski, 1998] Mikhajlov, L. y Sekerinski, E. (1998). A study of the fragile base class problem. En *12th European Conf. on Object-Oriented Programming (ECOOP'98)*, núm. 1445 de LNCS, pp. 355–382. Springer Verlag.

[Mikhajlova, 1999] Mikhajlova, A. (1999). *Ensuring Correctness of Object and Component Systems*. Tesis Doctoral, Åbo Akademi University.

[Milner, 1989] Milner, R. (1989). *Communication and Concurrency*. Prentice Hall.

[Milner, 1991] Milner, R. (1991). The Polyadic π-calculus: a Tutorial. Informe técnico, University of Edinburgh (Reino Unido).

[Milner, 1992] Milner, R. (1992). Functions as Processes. *Mathematical Structure in Computer Science*, 2(2):119–146.

[Milner et al., 1992] Milner, R., Parrow, J., y Walker, D. (1992). A Calculus of Mobile Processes. *Journal of Information and Computation*, 100:1–77.

[Milner et al., 1990] Milner, R., Tofte, M., y Harper, R. (1990). *The definition of Standard ML*. MIT Press.

[Moller y Tofts, 1990] Moller, F. y Tofts, C. (1990). A temporal calculus for communicating systems. En *CONCUR'90*, núm. 458 de LNCS, pp. 401–415. Springer Verlag.

[Moore et al., 1999] Moore, A. P., Klinker, J. E., y Mihelcic, D. M. (1999). How to Construct Formal Arguments that Persuade Certifiers. En *Industrial-Strength Formal Methods in Practice*. Springer Verlag.

[Nestmann y Pierce, 1996] Nestmann, U. y Pierce, B. C. (1996). Decoding Choice Encodings. En Montanari, U. (ed.), *CONCUR'96*, núm. 1119 de LNCS, pp. 179–194. Springer.

[Nierstrasz, 1995a] Nierstrasz, O. (1995a). Regular Types of Active Objects. En *Object-Oriented Software Composition*, pp. 99–121. Prentice Hall.

[Nierstrasz, 1995b] Nierstrasz, O. (1995b). Requirements for a Composition Language. En *ECOOP'94 Workshop on Models and Languages for Coordination of Parallelism and Distribution*, núm. 924 de LNCS, pp. 147–161. Springer Verlag.

[Nierstrasz y Dami, 1995] Nierstrasz, O. y Dami, L. (1995). Component-Oriented Software Technology. En Nierstrasz, O. y Tsichritzis, D. (eds.), *Object-Oriented Software Composition*, pp. 3–28. Prentice Hall.

[Nierstrasz et al., 1992] Nierstrasz, O., Gibbs, S., y Tsichritzis, D. (1992). Component-Oriented Software Development. *Communications of the ACM*, 35(9):160–165.

[Nierstrasz y Meijler, 1994] Nierstrasz, O. y Meijler, T. D. (1994). Requirements for a Composition Language. En *ECOOP'94 Workshop Reader*, núm. 924 de LNCS, pp. 147–161. Springer Verlag.

[Nierstrasz y Meijler, 1995] Nierstrasz, O. y Meijler, T. D. (1995). Research Directions in Software Composition. *ACM Computing Surveys*, 27(2):262–264.

[Odenthal y Quibeldey-Cirkel, 1997] Odenthal, G. y Quibeldey-Cirkel, K. (1997). Using Patterns for Design and Documentation. En *11th European Conf. on Object-Oriented Programming (ECOOP'97)*, núm. 1241 de LNCS, pp. 511–529. Springer Verlag.

[Olderog, 1991] Olderog, E. R. (1991). *Nets, Terms and Formulas*. Cambridge University Press.

[OMG, 1999] OMG (1999). *The Common Object Request Broker: Architecture and Specification*. Object Management Group, 2.3 edición. http://www.omg.org.

[OSF, 1994] OSF (1994). *OSF DCE Application Development Guide*. Open Software Foundation, Cambridge, MA (Estados Unidos). http://www.opengroup.org.

[Palamidessi, 1997] Palamidessi, C. (1997). Comparing the Expressive Power of the Synchronous and the Asynchronous π-calculus. En *24th ACM Symposium on Principles of Programming Languages (POPL'97)*, pp. 256–265. ACM Press.

[Pierce, 1997] Pierce, B. C. (1997). *Programming in the π-calculus. A tutorial introduction to Pict*. Indiana University, Computer Science Department, version 4.0 edición.

[Pierce y Turner, 1999] Pierce, B. C. y Turner, D. (1999). Pict: A Programming Language Based on the π-calculus. En *Proof, Language and Interaction*. MIT Press.

[Pistore y Sangiorgi, 1996] Pistore, M. y Sangiorgi, D. (1996). A Partition Refinement Algorithm for the π-calculus. En *Computer Aided Verification CAV'96*, núm. 1102 de LNCS, pp. 38–49. Springer Verlag.

[Pree, 1996] Pree, W. (1996). *Framework Patterns*. SIGS Publications.

[Prieto-Díaz y Neighbors, 1986] Prieto-Díaz, R. y Neighbors, J. M. (1986). Module Interconnection Languages. *Journal of Systems and Software*, 6(4):307–334.

[Radestock y Eisenbach, 1994] Radestock, M. y Eisenbach, S. (1994). What Do You Get From a π-calculus Semantics? En *PARLE'94*, núm. 817 de LNCS, pp. 635–647. Springer Verlag.

[RAISE, 1992] RAISE (1992). *The RAISE Specification Language*. Prentice Hall.

[RAISE, 1995] RAISE (1995). *The RAISE Development Method*. Prentice Hall.

[Reed y Roscoe, 1988] Reed, G. M. y Roscoe, A. W. (1988). A timed model for communicating sequential processes. *Theoretical Computer Science*, 1(58):249–261.

[Rogerson, 1997] Rogerson, D. (1997). *Inside COM*. Microsoft Press.

[Rumbaugh et al., 1991] Rumbaugh, J. et al. (1991). *Object-Oriented Modeling and Design*. Prentice Hall.

[Rumbaugh et al., 1999] Rumbaugh, J. et al. (1999). *The Unified Modeling Language Reference Manual*. Addison Wesley.

[Sangiorgi, 1992] Sangiorgi, D. (1992). The Lazy Lambda Calculus in a Concurrency Scenario. En *LICS'92*. IEEE.

[Sangiorgi, 1993] Sangiorgi, D. (1993). A theory of bisimulation for the π-calculus. Informe Técnico ECS-LFCS-93-270, University of Edinburgh (Reino Unido).

[Saraswat y Rinard, 1991] Saraswat, V. y Rinard, M. (1991). Concurrent Constraint Programming. En *17th ACM Symposium on Principles of Programming Languages*. ACM Press.

[Schmid, 1997] Schmid, H. A. (1997). Systematic Framework Design by Generalization. *Communications of the ACM*, 40(10):48–51.

[SEI, 1990] SEI (ed.) (1990). *Workshop on Domain-Specific Software Architectures*. Software Engineering Institute.

[Shaw, 1995] Shaw, M. (1995). Architectural Issues in Software Reuse. En *SIGSOFT Symposium on Software Reusability*, pp. 3–6. ACM Press.

[Shaw, 1999] Shaw, M. (1999). Architectural Challenges in Integration and Interoperability. Informe técnico, Carnegie-Mellon University, (Estados Unidos).

[Shaw et al., 1995] Shaw, M. et al. (1995). Abstractions for Software Architecture and Tools to Support Them. *IEEE Trans. on Software Engineering*, 21(4):314–335.

[Shaw y Garlan, 1995] Shaw, M. y Garlan, D. (1995). Formulations and Formalisms in Software Architecture. En van Leeuwen, J. (ed.), *Computer Science Today*, núm. 1000 de LNCS, pp. 307–323. Springer Verlag.

[Shaw y Garlan, 1996] Shaw, M. y Garlan, D. (1996). *Software Architecture. Perspectives of an Emerging Discipline*. Prentice Hall.

[Singhal, 1989] Singhal, M. (1989). Deadlock Detection in Distributed Systems. *IEEE Computer*, 22:37–48.

[Spivey, 1992] Spivey, J. M. (1992). *The Z Notation. A Reference Manual*. Prentice Hall.

[Sun Microsystems, 1997] Sun Microsystems (1997). JavaBeans API Specification 1.01.

[Szyperski, 1998] Szyperski, C. (1998). *Component Software: Beyond Object-Oriented Programming*. Addison Wesley.

[Szyperski, 2000] Szyperski, C. (2000). Components and Architecture. *Software Development Online*, (10). http://www.sdmagazine.com.

[Szyperski y Pfister, 1996] Szyperski, C. y Pfister, C. (1996). Summary of the Workshop on Component Oriented Programming (WCOP'96). En Mühlhäuser, M. (ed.), *ECOOP'96 Workshop Reader*. Dpunkt Verlag.

[Toval y Fernández-Alemán, 2000] Toval, A. y Fernández-Alemán, J. L. (2000). Formally Modeling UML and its Evolution: a Holistic Approach. En *Formal Methods for Open Object-Based Distributed Systems (FMOODS 2000)*. Kluwer Academic Publishers.

[Toval y Fernández-Alemán, 2001] Toval, A. y Fernández-Alemán, J. L. (2001). Improving System Reliability via Rigorous Software Modeling: The UML Case. En *IEEE Aerospace Conference'2001*, Montana (Estados Unidos). (Aceptado).

[Vallecillo, 1999] Vallecillo, A. (1999). *Un modelo de componentes para el desarrollo de aplicaciones distribuidas*. Tesis Doctoral, Universidad de Málaga, Departamento de Lenguajes y Ciencias de la Computación.

[Vera et al., 1999] Vera, J., Perrochon, L., y Luckham, D. C. (1999). Event-Based Execution Arquitectures for Dynamic Softweare Systems. En *Software Architecture*, pp. 303–317. Kluwer Academic Publishers.

[Vestal, 1993] Vestal, S. (1993). An Overview and Comparison of Four Architecture Description Languages. Informe técnico, Honeywell Technology Center.

[Victor, 1994] Victor, B. (1994). A Verification Tool for the Polyadic π-calculus. Tesis de licenciatura, Department of Computer Systems, Uppsala University.

[Vinoski, 1998] Vinoski, S. (1998). New Features for CORBA 3.0. *Communications of the ACM*, 41(10):44–52.

[Viry, 1996a] Viry, P. (1996a). Input/Output for ELAN. *Electronic Notes in Theoretical Computer Science*, 4.

[Viry, 1996b] Viry, P. (1996b). A Rewriting Implementation of π-calculus. Informe Técnico TR-96-29, Università di Pisa, Dipartimento de Informatica, Pisa, (Italia).

[Walker, 1991] Walker, D. (1991). π-calculus Semantics of Object-Oriented Programming Languages. En *Theoretical Aspects od Computer Science*, núm. 526 de LNCS, pp. 532–547. Springer Verlag.

[Walker, 1995] Walker, D. (1995). Objects in the π-calculus. *Journal of Information and Computation*, 116(2):253–271.

[Wallnau et al., 1997] Wallnau, K. et al. (1997). Engineering Component-Bases Systems with Distributed Object Technology. En *World-Wide Computing and Its Applications (WWCA'97)*, núm. 1274 de LNCS, pp. 58–73. Springer Verlag.

[Whittle, 2000] Whittle, J. (2000). Formal approaches to systems analysis using UML: A survey. *Journal of Database Management*, 1(4):4–13.

[Yellin y Strom, 1997] Yellin, D. M. y Strom, R. E. (1997). Protocol Specifications and Components Adaptors. *ACM Trans. on Programming Languages and Systems*, 19(2):292–333.

[Zaremski y Wing, 1997] Zaremski, A. y Wing, J. (1997). Specification Matching of Software Components. *ACM Trans. on Software Engineering and Methodology*, 6(4):333–369.