Un Modelo de Adición de Propiedades a Componentes Software

Antonio Vallecillo Moreno
Dpto. de Lenguajes y Ciencias de la Computación.
Universidad de Málaga. Campus de Teatinos, s/n.
29071 Málaga, España.
E-mail: av@lcc.uma.es

Abstract

Dentro de las metodologías de diseño de software, la programación orientada a componentes (COP) comienza a mostrarse como una extensión natural de la programación orientada a objetos (OOP) que potencia la reusabilidad del software. Además, el diseño de software para sistemas abiertos y distribuidos se enfrenta a problemas complejos y particulares como son la dispersión física de los componentes y su adaptación a entornos heterogéneos y cambiantes. El presente trabajo describe las habilidades que los componentes deben desarrollar para adaptarse a estos sistemas, así como un modelo general que permite añadir tales propiedades a componentes existentes. El modelo da lugar a dos resultados adicionales: por un lado una metodología general de diseño de componentes basados en la adición modular de propiedades; por otro, una visión distinta de la coordinación (autocoordinación), en donde son los propios componentes, y no el sistema, quienes se responsabilizan de conseguir coordinar a los componentes adecuados para cumplir su misión.

1 Introducción

Conforme aumenta el uso de los sistema abiertos y distribuidos cobra mayor importancia el estudio de los problemas que se plantean en ellos. Una de las características que diferencia a estos sistemas del resto es que las entidades computacionales que viven en ellos pueden ser consideradas componentes software encapsulados y reactivos, dispersos físicamente, y que interoperan en entornos heterogéneos y cambiantes [17, 11].

La programación orientada a componentes (COP) cobra forma como una extensión natural de la programación orientada a objetos (OOP) que permite describir mejor el comportamiento de tales sistemas y tratar los problemas que en ellos se plantean. La programación de estos sistemas puede ser considerada como una *coordinación* de tales componentes mientras éstos operan de forma concurrente.

En nuestro modelo cada entidad computacional es una *unidad individual*, con una tarea que cumplir, su *objetivo*. Para realizarlo ha de saber adaptarse al medio, utilizar los recursos que éste le ofrece y vender sus servicios. Pero al tratarse de un medio cambiante, el componente se enfrenta con problemas que dificultan su misión: nuevos servidores y servicios aparecen y otros desaparecen o dejan de funcionar; es necesario coordinarse con otros componentes cuyo interfaz es desconocido; por último, los errores, retrasos y fallos en la comunicación no pueden ser ignorados.

Además, el hacer frente a esos nuevos retos no debería complicar innecesariamente el núcleo fundamental del componente. Aparece por tanto la necesidad de que los componentes desarrollen distintas habilidades como pueden ser la autonomía, adaptabilidad, robustez o seguridad, y que lo puedan hacer de forma modular. Se necesita entonces disponer de un esquema y una metodología que permita diseñar componentes con tales propiedades, y que también permita incorporar tales facultades a componentes ya existentes.

Aparte, hemos de tener en cuenta que en los sistemas abiertos y distribuidos deja de existir una visión global del sistema, siendo únicamente posible disponer de visiones locales del mismo desde los componentes. Esta concepción ha dirigido nuestro enfoque hacia estos últimos y no hacia el sistema entero. Para nosotros son los componentes los responsables de sí mismos, siendo ellos los encargados de saber adaptarse al medio, buscar aliados para conseguir su objetivo y ser capaces de coordinarse entre sí. Esto define el concepto de *Autocoordinación*.

La contribución de este artículo es doble. Por un lado se describen las propiedades básicas que permitirán a los componentes lograr su objetivo por sí mismos en sus entornos cambiantes, heterogéneos y distribuidos. Por otro, se define un esquema general para definir y añadir propiedades a los componentes de un sistema, de una forma modular e independiente. Tal modelo facilita la especificación, implementación y estudio de tales propiedades, a la vez que introduce una metodología general para el diseño de componentes.

A partir de aquí, la siguiente sección describe un conjunto de habilidades o propiedades para los componentes de estos sistemas. La sección 3 describe el modelo usado para definir propiedades y añadirlas a los componentes, y en la sección 4 se muestra un ejemplo de una de ellas (la Independencia). Finalmente las secciones 5 y 6 presentan los trabajos relacionados con este tema y algunas conclusiones.

2 Propiedades de los Componentes

El objetivo de esta sección es describir, de manera breve e informal, varias propiedades para los componentes de los sistemas abiertos y distribuidos. Vamos a definir tres fundamentales: la Autonomía, la Robustez y la Competitividad. Entendemos por *Autonomía* la habilidad de un componente de tomar decisiones de forma independiente. La *Robustez* garantiza la disponibilidad, fiabilidad y seguridad del componente. Y la *Competitividad* dota al componente de una estrategia general de supervivencia en términos de durabilidad y resistencia.

El problema es que estas tres propiedades son demasiados generales y complejas para ser implementadas directamente. La idea clave es "descomponerlas", definiéndolas en función de propiedades más básicas. Así vamos a definir la propiedad de **Autonomía** como la composición de otras tres propiedades:

- *Independencia*: El componente ha de disponer de autogobierno, ser capaz de buscar los servicios que necesita y de decidir libremente qué solución escoger entre las posibles.
- *Adaptabilidad*: El componente ha de ser capaz de acomodarse a distintos interfaces y protocolos, y a cambios en sus requerimientos. Debe poder componerse con otros y ser flexible, versátil, personalizable y extensible.
- *Autoprotección*: El componente debe protegerse ante fallos externos y circunstancias inesperadas, con el objeto de no depender nunca del comportamiento y buen funcionamiento del resto de los componentes.

La **Robustez** se consigue mediante la composición de tres propiedades básicas:

- *Integridad*: El componente ha de ofrecer un comportamiento robusto frente a distintos usos de su interfaz, tanto válidos como inválidos.
- *Control de Acceso*: Todos los accesos hacia (y desde) el componente deben ser autentificados y autorizados convenientemente.

- *Alta Disponibilidad* (High Availability): El componente ha de protegerse ante la ocurrencia de fallos en los procesos o máquinas en donde se ejecuta.

Finalmente, la **Competitividad** puede expresarse en función de las siguientes propiedades:

- *Mayor Esfuerzo/Menor Desgaste*: El componente debe hacer sus mayores esfuerzos para satisfacer las peticiones de sus usuarios (en términos de tiempo de respuesta, funcionalidad y calidad del servicio), mientras que debe minimizar sus costes (o sufrimientos) para realizarlo.
- *Durabilidad*: El componente debe incorporar mecanismos que le permitan actualizarse, renovarse y mejorar con el tiempo.

Nuestro objetivo ahora es definir un modelo general capaz de especificarlas de una manera formal e implementarlas, permitiendo añadirlas a los componentes de un sistema, y componerlas entre sí. El modelo general da lugar de manera natural a una metodología de diseño de componentes, en donde éstos pueden ser especificados de forma modular y progresiva: el componente puede concentrarse en sus labores meramente computacionales para luego "adquirir" las propiedades que crea necesarias, descargando su diseño de detalles ajenos a su objetivo.

En el presente trabajo sólo hemos definido las propiedades que consideramos básicas. En cualquier caso, nuestro modelo permite la especificación de nuevas propiedades, que pasarían a formar parte de las habilidades disponibles para todos los componentes.

3 El Modelo

Los componentes de los sistemas abiertos y distribuidos pueden ser vistos como encapsulaciones de programas, las cuales abstraen sus propiedades, ofrecen un interfaz común para acceder a sus servicios, y permiten ocultar su implementación, componerlos y coordinarlos. Esto ocurre además independientemente del paradigma de programación escogido, ya sea mediante objetos, actores, lenguajes de coordinación, etc.

Pues bien, la idea común es la de ir incorporando a esas "cápsulas" nuevas *capas* que añadan la propiedad escogida en cada caso. Este es un mecanismo conocido y aceptado [8, 3], y buenos ejemplos son las arquitecturas regidas por leyes para Linda [14], el control de tiempo real para los actores [16] y objetos [17], o el modelo de filtros [2].

Cada una de esas capas actúa como un envolvente (*wrapper*) [8] activo que captura y modifica la entrada y salida de datos del programa, y lo dota de la funcionalidad pedida. Sin embargo, las capas a las que nos referimos aquí no se comportan como simples filtros, pues su comportamiento no es meramente computacional o pasivo: como resultado de la llegada de un mensaje, la capa puede enviar uno o más mensajes al sistema, esperar sus respuestas y construir con ellas el mensaje final que pasará al componente. Además, estas capas pueden "componerse" funcionalmente para ir modelando el componente a nuestro gusto.

Una gran ventaja de este enfoque es la independencia que se consigue entre el programa y cada una de las capas, pues cada una realiza su función de forma independiente e incremental, permitiendo así una gran modularidad y flexibilidad. También permite "descargar" al componente de aquellas funciones que no son propiamente suyas, y ayuda a resolver de forma natural el problema de los componentes antiguos (*legacy systems*). Por supuesto, las propiedades no tienen porqué implementarse en todos los componentes de un sistema, ni suponer que todos han de cumplirlas; según nuestra filosofía, cada componente es libre de escoger su propia configuración.

3.1 El Modelo Teórico

La siguiente definición muestra los elementos que caracterizan a una propiedad $\mathcal{P}$ cualquiera.

Definición 1 *Una propiedad $\mathcal{P}$ definida en un Sistema S está caracterizada por una cuaterna*

$$\mathcal{P} = (\mathcal{C}, \mathcal{B}, \mathcal{E}, \mathcal{R})$$

donde $\mathcal{C}$ es un conjunto de Componentes de S que implementan esta propiedad; $\mathcal{B}$ es un conjunto de Controladores, habiendo uno asociado a cada componente de $\mathcal{C}$; $\mathcal{E}$ es un conjunto de Eventos de comunicación relativos a la propiedad $\mathcal{P}$; y $\mathcal{R}$ es el conjunto de reglas o Estrategias que definen los posibles comportamiento de los controladores de $\mathcal{B}$ ante los eventos de $\mathcal{E}$.

En esta definición, los *Controladores* son los procesos que implementan las capas. Ellos capturan todos los mensajes entrantes y salientes del componente (los *Eventos*, ya sean mensajes, RPCs o accesos a pizarras compartidas) y los modifican según la *Estrategia* que implementen. Las estrategias se definen y especifican a nivel global cuando se define una propiedad, y cada controlador escoge la estrategia apropiada cuando se particulariza para un componente concreto. El conjunto $\mathcal{R}$ contiene a todas las posibles estrategias para todos los eventos relativos a la propiedad.

En general, las propiedades no controlan la ocurrencia de los eventos, sino que sólo toman decisiones sobre lo que hacer con ellos. Y por supuesto, dos o más propiedades pueden componerse funcionalmente para formar nuevas propiedades bajo este esquema. La operación de composición permite encadenar unas propiedades a otras, y dota al conjunto $\{\mathcal{P}\}$ de todas la propiedades de estructura de monoide no conmutativo [18].

3.2 El Modelo Computacional

Para desarrollar el modelo teórico utilizaremos un modelo computacional que permita construirlo. El modelo escogido es muy simple y general, pues incorpora sólo la funcionalidad necesaria para nuestros propósitos. El objetivo de esto es poder implementarlo sobre cualquier tipo de sistemas y paradigmas (Linda [4], Actores [1], CORBA [11], Infospheres [6]).

Nuestro modelo computacional está basado en procesos que se comunican a través de colas de mensajes (buzones). Cada componente es un proceso (posiblemente multi-hebra) que tiene estados. Cada estado está compuesto por un conjunto de asignaciones de valores a las variables definidas en tal componente. La comunicación entre componentes es asíncrona, y se realiza mediante el envío mensajes a los buzones.

Para cada componente existe definido su *dominio*, que no es sino la dirección de la máquina (o red de máquinas) en donde decide establecerse. Como nombres de dominio vamos a considerar los nombres de dominio de correo Internet, precedidos por el carácter "`@`". Ejemplos son "`@lcc.uma.es`" o "`@hal.com`".

Cada buzón tiene una dirección única global, que viene dada por una expresión de la forma "`mb@domain`", siendo "`mb`" el nombre asociado a tal buzón, que debe ser único dentro del dominio "`@domain`". Si se omite el dominio, por defecto se utiliza el del proceso que referencia al buzón. Si sólo se indica el dominio, se hace referencia a *todos* los buzones que pertenecen a él en ese momento. Esto permite realizar la difusión de mensajes (envíos *broadcast* a dominios).

En nuestro modelo cada componente tiene asociado un buzón, y existen dos operaciones básicas de comunicación: `Send` y `Receive`. La primera envía un mensaje a un buzón dado, y la segunda permite al componente leer un mensaje de su buzón. El envío de mensajes no es bloqueante pero sí la recepción, en caso de que el buzón esté vacío.

Los mensajes están compuestos por campos, conteniendo entre otros la dirección del buzón destino (`m.To`), la dirección del buzón origen (`m.Reply`), el "subject" (`m.Subj`) o campo de información libre que puede ser utilizado para resumir el objetivo del mensaje, un número de referencia del mensaje (`m.Ref`) y por último la información propia del mensaje (`m.Info`).

Es importante observar que el uso de este modelo permite aplicar también métodos de verificación y razonamientos formales [5] tanto sobre los componentes como sobre sus composiciones.

3.3 Los Controladores

En este modelo los controladores son, como hemos mencionado anteriormente, procesos especiales que *envuelven* al componente modificando su comportamiento. Ellos capturan los eventos entrantes y salientes del componente mediante accesos a sus colas, y los tratan de acuerdo a la estrategia que implementen. Esta idea tiene analogía con los decoradores o adaptadores para objetos [8], con el modelo reflectivo para actores [1], y con el modelo de filtros [2].

Para especificar los controladores vamos a utilizar un esquema genérico, que iremos derivando o *especializando* progresivamente: primero para especificar los controladores que implementan cada propiedad, y luego cada uno de estos habrá de especializarse en el momento de particularizarse a trabajar con un componente concreto.

Para especificar los controladores es necesario definir los tipos de mensajes que aceptan y las operaciones que soportan para manejarlos. Tres son las operaciones que serán invocadas por el sistema cuando se produzcan determinados eventos: `Deliver`, `Received` y `TimeoutExp`.

La primera permite capturar los mensajes salientes, y será invocada por el sistema cada vez que el componente desee enviar un mensaje a un buzón. La segunda función permite al controlador capturar los mensajes entrantes, y será invocada por ell sistema cada vez que se intente encolar un mensaje en la cola de mensajes asociada al controlador. Por último, la función `TimeoutExp` permitirá al componente conocer cuándo un mensaje que se envía no recibe respuesta; si bien es cierto que una propiedad cualquiera no tiene porqué establecer timeouts, en general los controladores sí deben saber tratar estas condiciones excepcionales, que no pueden olvidarse en los sistemas abiertos y distribuidos.

El sistema ofrece dos operaciones a los controladores para poder completar sus acciones: una vez tratado un mensaje entrante, el controlador lo depositará en la cola de mensajes del componente (`Inq`). Análogamente para mensajes salientes, el controlador añadirá a la cola del componente (`Outq`) el resultado del tratamiento que ha realizado sobre el mensaje. Con todo esto, el esquema más simple que podemos tener de un controlador es el siguiente[1]:

```
public class Controller {
  ...
  public void Deliver(Msg m) { // Captura mensajes salientes
      Outq.Queue(m);       // y los deja pasar sin modificarlos.
  }
  public void Received(Msg m) { // Captura mensajes entrantes
      Inq.Queue(m);        // y los pasa al componente directamente.
  }
  public void TimeoutExp(Msg m) { // Captura condiciones de timeout.
  }                        // No hace nada al respecto.
}
```

A partir de este esquema general iremos "derivando" los controladores que implementan cada propiedad mediante sucesivas especializaciones.

Obsérvese que en el caso de tener varios controladores enlazados consecutivamente (por

[1]Hemos utilizado Java [15] como lenguaje para especificar e implementar nuestro modelo.

tratarse de una composición de propiedades), el fin de cada operación `Inq.Queue` desencadenará una operación `Received` en el siguiente controlador y, de igual forma, el fin de cada operación `Outq.Queue` hará que se invoque a la función `Deliver` del siguiente controlador. En cuanto a la ocurrencia de timeouts, la función que realiza su tratamiento será invocada en todos los controladores asociados a un componente dado.

4 La propiedad de Independencia

Los sistemas abiertos y distribuidos son, en muchos aspectos, un fiel reflejo de nuestro mundo real. Los servicios que necesitamos no están siempre disponibles, aparecen nuevos servidores de vez en cuando, y otros desaparecen. Nosotros funcionamos con dos tipos de informaciones: las de confianza o *conocidas*, y las existentes en el mercado o *desconocidas*. Las primeras son las que hemos utilizado para resolver problemas que hemos tenido. De la existencia de las segundas nos enteramos a través de preguntas que hacemos o de publicidad que recibimos. Cuando necesitamos algún servicio usamos una solución conocida, a menos que no funcione en ese instante o hayamos oído de otra mucho mejor. Incluso si no conocemos ninguna ruta alternativa, sabemos que "preguntando se va a Roma". La facultad de buscar los servicios que necesitamos, junto con la capacidad de decidir cuál escoger en cada momento, nos permite tener *Independencia*. En este modo de funcionar nos apoyaremos para especificar los controladores de esta propiedad.

De acuerdo a nuestro modelo teórico, lo que sigue es una descripción de los elementos que componen la cuaterna que caracteriza la propiedad $\mathcal{I} = (\mathcal{C}, \mathcal{B}, \mathcal{E}, \mathcal{R})$.

4.1 Eventos Relativos a la Propiedad de Independencia

Los eventos relativos a esta propiedad van a ser todos aquellos mensajes de petición de prestación de servicio a otros componentes del sistema, junto con la publicidad recibida sobre nuevos servicios. Las peticiones de prestación de servicio recibidas por el componente no se consideran eventos relativos a esta propiedad.

4.2 Las Estrategias de $\mathcal{I}$

Son las diferentes formas en las que el controlador de un proceso toma decisiones sobre los servicios: cómo y a quién contratarlos y cómo mantener actualizada la información disponible.

Cara a tomar una decisión, podemos seguir muy distintas estrategias según el carácter que queramos imprimir a nuestro componente. Así, un programa conservador intentará una solución (esto es, un par servicio/proveedor) conocida siempre que pueda. Sólo decidirá cambiar de solución cuando ésta ya no le valga, haya dejado de funcionar, o exista otra de la que ha "oído" que la supera con creces. Un programa más atrevido escogerá la opción que sea más atractiva entre todas las que conoce o ha oído hablar, no dando demasiado peso a su experiencia anterior. Con la publicidad también existen varias alternativas: o bien se tira toda, y cuando no dispongamos de solución conocida entonces preguntamos a los componentes de nuestro dominio; o bien se va leyendo todo lo que llegue y actualizando nuestra lista de soluciones conocidas.

4.3 Controladores o "Brokers"

En el caso de la Independencia, los controladores van a ser los encargados de buscar, gestionar y administrar los servicios que el componente necesita, y por tanto también los denominaremos *Brokers*. Al descargar estas labores en los controladores conseguimos que el componente sea independiente frente a los cambios en su entorno. El uso de un controlador individual para cada componente permite su particularización y especialización de acuerdo a sus preferencias, y evita la dependencia de un servidor centralizado o común para muchos componentes. Obsérvese que nosotros no estamos en contra de estos servidores centralizados; si existen, el controlador debe utilizarlos para evitar duplicidad de trabajos y responsabilidades. Sin embargo, pensamos que los controladores deben ser implementados a nivel individual para conseguir la deseada independencia.

Para entender el concepto de controlador respecto a la propiedad de Independencia, una analogía clara aparece cuando pensamos en algunos gestores de servicios de nuestro mundo real: asesores de inversiones, agencias de viajes o corredores inmobiliarios. Otra idea interesante es la de la secretaria eficiente, que se ocupa de los vuelos y alojamientos de su jefe. Ella conoce sus preferencias y está al tanto de descuentos en hoteles, ofertas de viajes y huelgas aéreas para conseguir siempre las mejores combinaciones y precios, mientras que él puede concentrarse plenamente en sus reuniones.

Para llevar a cabo su misión, los controladores implementan los siguientes mecanismos: a) Mantenimiento de una lista ponderada de *Soluciones* conocidas; b) *Preguntas* a todos los componentes (broadcast); y c) Recepción y distribución de *Publicidad* asíncrona.

El primero de ellos consiste en una lista de los servicios que el controlador conoce. Esta lista se mantiene en torno a una función que asocia a cada solución un peso (o coste) y que permitirá al controlador comparar soluciones y tomar decisiones sobre el servicio a contratar.

Respecto a las preguntas difundidas a todos los componentes, éstas son necesarias cuando el componente pide al controlador algún servicio no conocido por éste, aunque también pueden ser utilizadas por el controlador para actualizar periódicamente su lista de soluciones. El sentido en el que utilizamos aquí la palabra difusión (*broadcast*) no implica que los mensajes han de llegar necesariamente a todos los componentes del sistema, ni que todos tengan que responder. De hecho, el componente no sabe cuántos otros componentes hay en un momento dado, ni tampoco le interesa ese dato. El sentido que le damos a esa palabra es más bien: "*a quien me escuche, me pueda entender, y esté interesado*".

Por otro lado, el concepto de publicidad aparece como la emisión y recepción de mensajes asíncronos que permiten estar al tanto de las novedades y cambios que se producen en el sistema. Cuando se recibe una publicidad se compara con la información que tenemos y se le asigna un peso, pudiendo incorporarse a la lista de soluciones mencionada anteriormente.

Obsérvese que no se trata sólo de la correduría (*trading*) de los servicios [10], sino que la idea clave es aquí la capacidad de decisión necesaria (en forma de función de peso) que posee el controlador para elegir siempre la mejor opción.

4.4 Un Ejemplo de Especificación de Controladores

El presente ejemplo muestra la implementación de un controlador de la propiedad de independencia para un componente dado `C`. A dicho controlador lo denominaremos $B_{\mathcal{I}}$.

Imaginemos que nuestro componente `C`, que vive en el dominio `@dom`, quiere viajar de A a B el 31/12/1999. Usando nuestro modelo, enviará un mensaje a su agencia de viajes usual `myTA@dom`:

```
msg.To    = "myTA@dom";
msg.Subj  = "ServiceRequest: getflightticket";
msg.Info().Serviceid    = "getflightticket";
msg.Info().Params().from  = "A";
msg.Info().Params().to    = "B";
msg.Info().Params().when  = new Date(1999,12,31);
msg.Info().Params().ccnum = "XX99XX99XX99XX99";
```

Si `C` incorpora la Independencia, el mensaje será interceptado por $B_{\mathcal{I}}$ antes de salir:

a) Si el proveedor `myTA@dom` está en su lista de soluciones conocidas y con el mayor peso, $B_{\mathcal{I}}$ dejará pasar el mensaje.

b) Si `myTA@dom` es conocido pero no tiene el mayor peso, dependiendo de su estrategia $B_{\mathcal{I}}$ puede: *i*) dejar pasar el mensaje (estrategia conservadora); *ii*) dejarlo pasar sólo si la diferencia entre los pesos de `myTA@dom` con el mejor de la lista es pequeña; o *iii*) reemplazar `myTA@dom` con el mejor de la lista directamente (estrategia liberal).

c) Si `myTA@dom` no fuera conocido, $B_{\mathcal{I}}$ lo reemplazará con el mejor de la lista. Si ésta estuviera vacía, $B_{\mathcal{I}}$ sencillamente dejaría pasar el mensaje.

Esto queda reflejado en la implementación de la función `Deliver` del controlador $B_{\mathcal{I}}$:

```
bestprovider = TheList.Best(msg.Info().Serviceid,msg.Info().Params());
if (!bestprovider.equals("")) {
  w1 = TheList.Weight(bestprovider,msg.Info().Serviceid,msg.Info().Params());
  w2 = TheList.Weight(msg.To,msg.Info().Serviceid,msg.Info().Params());
  if ((w1-w2)>0.2) msg.To = bestprovider;
};
Outq.Queue(msg);
```

Una vez el agente destino ha recibido el mensaje y efectuado la reserva, contestará a `C` con el resultado del servicio. Esta respuesta será interceptada por $B_{\mathcal{I}}$, que actualizará su lista con ella:

```
void Received(Msg m){ //captura mensajes entrantes
    TheList.Update(m.Reply,alive,m.Info().Serviceid,m.Info().Params());
    Inq.Queue(m);
};
```

Si la respuesta no llegara nunca y el componente `C` incorpora la propiedad de Autoprotección, el sistema invocará a su procedimiento de timeouts, que entre otras cosas permitirá al controlador registrar este hecho

```
TheList.Update(m.To,outoforder,m.Info().Serviceid,m.Info().Params());
```

Por otro lado, veamos lo que ocurre cuando el servidor `myTA@dom` no se encuentra disponible (está fuera de servicio o su dirección no es ya válida). En ese caso $B_{\mathcal{I}}$ necesita encontrar otro servidor para `C`. No hay problema si la lista de soluciones no es vacía, pero si no, la función `Deliver` debe incorporar el siguiente mecanismo:

```
bestprovider = TheList.Best(msg.Info().Serviceid,msg.Info().Params());
if (!bestprovider.equals("")) m.To = bestprovider;
else {
  msg.To    = ""; // broadcast a todos los componentes del dominio
  msg.Subj  = "InfoRequest: "+m.Info().Serviceid;
  msg.setInfo(m.Info());
  msgid = Mbox.Send(msg);
  do { rep = Mbox.Receive();
       TheList.Update(rep.Reply,alive,rep.Info().Serviceid,rep.Info().Params());
  } while (rep.RefTo != msgid);
  m.To = TheList.Best(m.Info().Serviceid,m.Info().Params());
};
Outq.Queue(m);
```

Obsérvese que en este ejemplo tomamos el primer proveedor que contesta a nuestra petición. Otras estrategias pueden decidir esperar tres respuestas y escoger la mejor, o bien esperar durante un tiempo (si el sistema lo incorpora) y escoger el mejor hasta el momento. Por último, sólo mencionar que si `C` no especificara el proveedor en su mensaje (`msg.To.equals("")`), $B_{\mathcal{I}}$ dejaría pasar el mensaje sin modificar.

5 Otros Trabajos Relacionados

La idea de este trabajo surge a partir de las ideas originales de Tokoro y Takashio [17], y de Minsky y Leichter [14]. Los primeros citan el tiempo real, la asincronía y la autonomía como piezas claves en los sistemas abiertos y distribuidos, pero sólo tratan en su trabajo los dos primeros conceptos, ignorando la autonomía. Por otro lado en [14] se presentan las arquitecturas regidas por leyes, esquema mono-propiedad que hemos generalizado y ampliado para dar lugar a nuestro modelo general de propiedades.

Respecto a los modelos de capas, éstos son muy conocidos y aceptados. Además de los "Home Processes" para Linda y el control de tiempo real para objetos de los dos trabajos anteriores, también citaremos el control de tiempo para actores [16] y el modelo de filtros [2]. Sin embargo, estas capas suelen ser meros filtros computacionales, a lo más con capacidad de retardo. Nuestras capas pasan a ser procesos más activos, con capacidad para interrogar a otros elementos del sistema y tomar decisiones dependiendo de las preferencias del componente, más al estilo de los adaptadores de [8]. Este modelo de procesos que capturan y modifican los eventos de comunicación de sus componentes es completamente reflectivo, al igual que los meta-actores [1] o los filtros [2]. Una buena descripción de modelos reflectivos puede encontrarse también en [13].

En cuanto a las propiedades definidas para los componentes, existe abundante bibliografía en el campo de la Inteligencia Artificial, en donde se discute acaloradamente sobre las posibles definiciones de agentes autónomos y de las propiedades que éstos deben cumplir [7]. Sin embargo, aparte de definiciones informales no conocemos ningún trabajo en este campo que, como nosotros, las defina y especifique formalmente.

El modelo computacional de procesos y buzones escogido no trata de ser novedoso, sino más bien simple y general con el objeto de poder implementarlo sin dificultad sobre cualquier otro modelo o paradigma computacional, como puede ser Linda, CORBA, Actores o Infospheres. El concepto de difusión de mensajes (broadcast) aquí incorporado generaliza las ideas de Gehani [9] y de espacios de actores [1].

Y en cuanto a la propiedad de Independencia y la gestión y administración de los servicios que un componente necesita, el concepto de correduría de servicios (*trading*) [10] es conocido e incluso forma parte de CORBA [12]. Sin embargo, esta propiedad no sólo trata de eso, sino que su importancia reside en la capacidad de decisión (en forma de función de peso) que poseen los controladores. De hecho, los controladores de esta propiedad implementados sobre CORBA deben hacer uso de las funciones de correduría de servicios que les ofrece el sistema, limitándose a la toma de decisiones.

El resto de las propiedades no se ha incluido aquí por cuestión de espacio. Todas ellas son de alguna forma conocidas y ya existen para distintos sistemas. El modelo aquí presentado ha demostrado su potencia al permitir especificarlas e implementarlas de forma unificada siguiendo nuestro esquema.

6 Conclusiones y Líneas de Trabajo

Hemos definido en el presente trabajo un conjunto de propiedades para los componentes software de los sistemas abiertos y distribuidos, junto con un modelo general que permite su definición formal, especificación e implementación, todo de forma modular. Usando este modelo se ha presentado una de las propiedades, la Independencia.

El disponer de un modelo general no sólo ofrece una plataforma para definir formalmente las propiedades mencionadas, sino que permite razonar sobre ellas e incorporar nuevas propiedades fácilmente. Además, define una metodología para el diseño de componentes.

Actualmente ya se han especificado las propiedades definidas aquí siguiendo nuestro esquema, quedando sólo pendiente alguna de sus posibles estrategias. Ahora trabajamos en varios frentes: por un lado, la definición de un marco de trabajo (*framework*) para el modelo completo; por otro, en una especificación formal de las estrategias que permita el desarrollo automático de prototipos para los esquemas de los controladores; y finalmente, en la implementación del modelo en varios sistemas.

References

[1] G. Agha, W. Kim and R. Panwar. *Actor Languages for Specification of Parallel Computations*. In DIMACS, 1994.

[2] L. Bergmans and M. Aksit. *Composing Synchronization and Real-Time Constraints*. In JPDC, No. 36, pp.32–52, Academic Press Inc., 1996.

[3] J. Bosch. *Language Support for Design Patterns*. In proceedings TOOLS Europe'96, 1996.

[4] N. Carriero and D. Gelernter. *Data Parallelism and Linda*. In LNCS, No. 757, pp. 145–159, Springer-Verlag, 1992.

[5] K.M. Chandy and A. Rifkin. *Systematic Composition of Objects in Distributed Internet Applications: Processes and Sessions*. In Proceedings of the 30th HICSS, 1997.

[6] K.M. Chandy et al. *A Framework for Structured Distributed Object Computing*. CalTech. Submitted to Comms. of the ACM, 1997.

[7] S. Franklin and A. Graesser. *It is an Agent, or just a Program?: A Taxonomy for Autonomous Agents*. In Proc. of the 3rd International Workshop on Agent Theories, Architectures and Languages, Springer-Verlag, 1996.

[8] E. Gamma et al. *Design Patterns*. Addison-Wesley, 1995.

[9] N. Gehani. *Broadcasting Sequential Processes (BSP)*. In *Concurrent Programming*, edited by N. Gehani and A. McGettrick, Addison-Wesley, 1988.

[10] Y. Hoffner. *A Designer's Introduction to Trading*. APM Technical Report APM1387.01, Architecture Projects Management, APM Ltd., Cambridge, UK, 1994.

[11] ISO/IEC 10746–1 to 10746–4. *Reference Model of Open Distributed Processing*. Draft Rec. X.901 to X.904 ISO/IEC JTC1/SC21/WG7, May 1995.

[12] ISO/IEC DIS 13235. *Information Technology - ODP - ODP Trading Function - Editors draft DIS text*. Draft Rec. X.9tr ISO/IEC JTC1/SC21, February 1996.

[13] F. Manola. *MetaObject Protocol Concepts for a "RISC" Object Model*. Technical Report TR-0244-12-93-165, GTE Laboratories Inc, 1993.

[14] N. Minsky and J. Leichter. *Law-Governed Linda as a Coordination Model*. In LNCS, No. 924, pp.125–146, Springer-Verlag, 1995.

[15] P. Naughton. *The Java Handbook*. Osborne McGraw-Hill, 1996.

[16] S. Ren, G. Agha and M. Saito. *A Modular Approach for Programming Distributed Real-Time Systems*. In JPDC, special issue on Object-Oriented Real-Time Systems, 1996.

[17] M. Tokoro and K. Takashio. *Toward languages and Formal Systems for Distributed Computing*. In LNCS, No. 791, pp.93–110, Springer-Verlag, 1993.

[18] J.M. Troya y A. Vallecillo. *An Open System Model based on Self-Coordinating Components*. Technical Report ITI-97-3, Dpto. Lenguajes y Ciencias de la Computación, Univ. de Málaga, 1997.