Sobre la búsqueda y emparejamiento de componentes COTS con múltiples interfaces

Luis Iribarne[1] y Antonio Vallecillo[2]

[1] Dpto. Lenguajes y Computación. Universidad de Almería.
e-mail: liribarn@ualm.es
[2] Dpto. Lenguajes y Ciencias de la Computación. Universidad de Málaga.
e-mail: av@lcc.uma.es

Resumen Actualmente existe un interés creciente por los procesos de búsqueda y emparejamiento de componentes COTS para construir con ellos aplicaciones software. Sin embargo, estos procesos siempre se han planteado desde una perspectiva simplista, en donde los componentes presentan una sola interfaz con los servicios que estos ofrecen, y en donde el emparejamiento se produce 1 a 1. Este trabajo aborda una extensión de dichos estudios a un ámbito más general, en el cual los componentes ofrecen varias interfaces, como suele suceder en las aplicaciones reales. En este contexto, se extienden los típicos operadores de composición, reemplazabilidad y compatibilidad de componentes para trabajar con múltiples interfaces. También, se estudian soluciones a ciertos problemas que aparecen al construir aplicaciones con este tipo de componentes.

1 Introducción

En la última década, la *ingeniería del software basada en componentes* ha experimentado un fuerte avance gracias al desarrollo de un software reutilizable conocido como componentes '*commercial off-the-shelf*'(COTS) [6,15,18]. Dichos componentes están modificando poco a poco la forma de construir aplicaciones dentro de las organizaciones, pues en vez de *desarrollarlas*, ahora se trata de *ensamblarlas* a partir de componentes COTS ya existentes. El objetivo, una vez más, es tratar de reducir el tiempo y los costes a la hora de construir aplicaciones software, mejorando su fiabilidad y seguridad al (re)utilizar componentes software ya probados y depurados con anterioridad.

Estos objetivos hacen que la ingeniería del software se enfrente a nuevos retos y problemas, ya que esta clase de componentes obliga a un desarrollo ascendente (*bottom-up*) de los sistemas frente al tradicional descendente (*top-down*). Mientras que en este último los requisitos del sistema van siendo desglosados (refinados) sucesivamente en otros más simples hasta llegar a los componentes finales, en el desarrollo ascendente parte de los requisitos iniciales han de ser cubiertos por componentes ya existentes que se encuentran almacenados en *repositorios de componentes* [12]. Por tanto, la especificación de dichos componentes ha de ser contemplada desde las primeras fases del diseño de una aplicación [14].

El objetivo es estudiar la construcción de aplicaciones basadas en componentes COTS a partir de las especificaciones de los componentes que integran su arquitectura. Estas especificaciones, que describen de forma *abstracta* los requisitos que deben cumplir los componentes del sistema, difiere en cierta medida de aquellas especificaciones *concretas* de los componentes COTS disponibles en los repositorios. Tradicionalmente, cada componente ofrecía un solo servicio, y cada servicio requerido se especificaba por separado. Por ello, los procesos de selección y emparejamiento se realizaban de 1 a 1. Sin embargo, esto no es posible en ambientes COTS. Estos componentes son de grano grueso, integran diferentes servicios, y ofrecen múltiples interfaces. Pensemos por ejemplo en componentes como un navegador de Internet o un procesador de textos, que ofrecen distintos servicios como un editor de páginas Web, un corrector ortográfico, etc.

La misión del presente trabajo es estudiar los problemas que se plantean en este tipo de entornos –problemas como las *lagunas* o los *solapamientos* de interfaces– extendiendo los tradicionales operadores de reemplazabilidad y compatibilidad de componentes para el caso que ofrezcan más de una interfaz. Asimismo, no sólo se tendrán en cuenta los servicios ofrecidos por los componentes, sino también los que necesitan de otros para funcionar, siguiendo las tendencias actuales de la *programación orientada a componentes* [16].

El trabajo está estructurado en 6 secciones, la primera de ellas se corresponde con esta *Introducción*. En la sección 2 se presenta una visión general de las aplicaciones basadas en componentes COTS. En la sección 3 se hace la propuesta concreta –las estrategias de composición y sus problemas– que luego se ilustra con un ejemplo en la sección 4. Seguidamente, en la sección 5, se describen algunos de los trabajos relacionados que han motivado la presente contribución. Por último, en la sección 6, se describen las conclusiones y los trabajos futuros.

2 Software basado en componentes comerciales

En primer lugar, se adoptará la definición de Clemens Szyperski para definir un componente software [16]: "Un componente es una unidad de composición de aplicaciones software, que posee un conjunto de interfaces y un conjunto de requisitos, y que ha de poder ser desarrollado, adquirido, incorporado al sistema y compuesto con otros componentes de forma independiente, en tiempo y espacio".

El adjetivo COTS se refiere a un tipo particular de componente, caracterizado por "ser de índole comercial, generalmente de grano grueso y de bajo coste, que es adquirido, seleccionado, probado, validado e integrado por desarrolladores de un sistema software basado en componentes para satisfacer ciertas necesidades del sistema, a partir de unos requisitos específicos"[18].

Aunque el uso de los componentes COTS en los procesos de desarrollo tradicionales de la ingeniería del software aun no está muy extendido, las ventajas que se esperan de estos han hecho nacer numerosas líneas de investigación, que cubren desde los primeros pasos del análisis de requisitos [14] hasta el estudio en tiempo de ejecución de las aplicaciones o la interoperabilidad entre componentes heterogéneos en sistemas abiertos. Una de estas líneas de actuación se centra

en la definición de unas plantillas de especificación de componentes COTS que puedan ser utilizadas por los clientes y proveedores como un lenguaje común para: *seleccionar* los COTS que se ajustan a las especificaciones de uno buscado; *probar* los seleccionados para escoger el más adecuado; *validar* sus resultados de prueba y su conveniencia de ser incorporado en el subsistema (incompatibilidad con la arquitectura); *comprar* el producto según los patrones de especificación como contrato cliente/vendedor; por último, *integrar* el componente.

Otro aspecto importante es la diferencia existente entre dos clases de desarrollo software: el *centrado* y el *basado* en COTS [2,10,18]. El primero se refiere a los procesos y técnicas software para el desarrollo exclusivo de componentes COTS. Actualmente es el más utilizado por las empresas que desarrollan aplicaciones procurando que sus componentes puedan ser luego reutilizados en otras, básicamente de la misma empresa. El desarrollo *basado* se refiere a un desarrollo a partir de componentes COTS ya existentes con el fin de ensamblar aplicaciones, en vez de desarrollarlas. El segundo enfoque constituye el objetivo a estudiar.

3 Sobre la composición con interfaces múltiples

Una vez establecido lo que se entiende por "componente COTS" y el desarrollo de aplicaciones a partir de estos componentes, seguidamente se van a estudiar algunos problemas que pueden surgir bajo este tipo de ambientes.

3.1 Interfaces

En primer lugar, es necesario hablar de las interfaces. Por una interfaz se entiende: "una abstracción de un servicio, que define las operaciones proporcionadas por dicho servicio, pero no sus implementaciones".

En general, una interfaz puede presentarse de distintas maneras. Por ejemplo, en CORBA las interfaces de los objetos siguen un enfoque orientado a objetos, formadas por el conjunto de las variables y métodos del objeto accesibles de forma pública. En COM los objetos pueden tener más de una interfaz, cada una de ellas con las signaturas de las operaciones que admite el componente. Esto sucede también en el nuevo modelo de componentes de CORBA (CCM [13]), en donde además, se contemplan las operaciones que los objetos necesitan de otros.

Obsérvese, sin embargo, que esta información se queda simplemente a un nivel sintáctico, es decir, describe sólo los nombres de los métodos y los tipos de sus parámetros y del valor que devuelven. No obstante, se ha visto que esa información no es suficiente a la hora de construir aplicaciones [19,20]. Por ejemplo, en interoperabilidad entre objetos se distinguen tres niveles: las *signaturas*, citado anteriormente; los *protocolos*, en donde se examinan el orden parcial de las llamadas a los servicios de un objeto y el orden en el que éste invoca los servicios externos de otros objetos; y finalmente el nivel *semántico*, para el significado de las operaciones y la especificación de su comportamiento [17]. En este sentido, la información que puede contener una interfaz va a depender del nivel con el que se pretenda describir.

A nivel de *signaturas*, una interfaz contiene sólo información sintáctica de las signaturas de las operaciones entrantes y salientes de un componente. Esta información puede completarse con los *protocolos* de interacción del componente. Para lo cual existen diferentes propuestas –dependiendo del formalismo para describir los protocolos– como máquinas de estados finitas [19], redes de Petri [3], lógica temporal [8,9] o el π-cálculo [4]. En este caso, a las interfaces se las denomina *roles*. La información a nivel *semántico* describe el comportamiento de las operaciones, con formalismos que van desde las pre/post condiciones e invariantes (p.e. Larch [5,20]), las ecuaciones algebraicas [7] o el cálculo de refinamiento [11].

3.2 Operaciones con interfaces

Independientemente de la información que puedan contener las interfaces, es necesario tener definido un conjunto de *operadores* que trabajen sobre ellas. Por ejemplo, uno de estos operadores determina cuándo una interfaz puede sustituir a otra sin que sus clientes sean capaces de apreciar el cambio. Este es el operador de *reemplazabilidad* entre interfaces, denotado por "$\sqsubseteq$". A nivel de signaturas este operador sólo exige que los métodos del primero estén contenidos en los del segundo. A nivel de semántica operacional este operador se conoce como subtipado de comportamiento (*behavioral subtyping*) [1]. Basándose en este operador —independientemente del nivel al que se defina— es posible definir una relación de equivalencia entre interfaces para decir que dos interfaces R_1 y R_2 son *equivalentes* (y se denota por $R_1 \equiv R_2$) sii $R_1 \sqsubseteq R_2$ y $R_2 \sqsubseteq R_1$.

Estos operadores son los que luego van a utilizar los *traders* de servicios para localizar en los repositorios software, los componentes que cumplen las especificaciones abstractas de los componentes definidos en la arquitectura de la aplicación. En la literatura, los operadores entre interfaces han sido definidos para interfaces simples. Nuestro propósito es extenderlos al caso de componentes con múltiples interfaces.

Definición 1 (Componente 'off-the-shelf') *Un componente C queda determinado por dos conjuntos de interfaces $C = (\mathcal{R}, \overline{\mathcal{R}})$. El primero con las interfaces de los servicios que* ofrece *a los demás componentes, $\mathcal{R} = \{R_1, ..., R_n\}$, y el segundo con los que* requiere *de los demás, $\overline{\mathcal{R}} = \{\overline{R}_1, ..., \overline{R}_m\}$.*

En lo que sigue se va a utilizar la notación $C.\mathcal{R}$ y $C.\overline{\mathcal{R}}$ para referirse a los conjuntos de interfaces del componente C. Por otro lado, los componentes se pueden *componer* para formar otros nuevos y otras aplicaciones.

Definición 2 (Composición de componentes) *Dados dos componentes $C_1 = (\mathcal{R}_1, \overline{\mathcal{R}}_1)$ y $C_2 = (\mathcal{R}_2, \overline{\mathcal{R}}_2)$, se define la composición de dos componentes $C_1 \mid C_2$ como un nuevo componente C_3 tal que:*

$$C_3 = (\mathcal{R}_3, \overline{\mathcal{R}}_3) = \begin{cases} (\mathcal{R}_1 \cup \mathcal{R}_2, \overline{\mathcal{R}}_1 \cup \overline{\mathcal{R}}_2 - \{\mathcal{R}_1 \cup \mathcal{R}_2\}) & \textit{sii } \mathcal{R}_1 \cap \mathcal{R}_2 = \varnothing \\ \textit{indefinido} & \textit{sii } \mathcal{R}_1 \cap \mathcal{R}_2 \neq \varnothing \end{cases}$$

Para evitar los conflictos que puedan aparecer en una aplicación en la que dos de sus componentes ofrezcan un mismo servicio al resto, es necesario definir una operación que permita *ocultar* una interfaz de un componente.

Definición 3 (Ocultación) *Dado un componente* $C_1 = (\mathcal{R}_1, \overline{\mathcal{R}}_1)$*, se define la ocultación respecto a un conjunto de interfaces* $\mathcal{R}$ *como* $C_1 - \{\mathcal{R}\} = (\mathcal{R}_1 - \mathcal{R}, \overline{\mathcal{R}}_1)$.

Además, una aplicación se considera como un nuevo componente que debe de ser *cerrado*, es decir, los servicios requeridos por los componentes integrantes deben quedar cubiertos por otros servicios que ofrecen sus propios componentes.

Definición 4 (Cierre) *Sea* C_1*,* C_2*, ...,* C_n *un conjunto de componentes y* $A = C_1 \mid C_2 \mid ... \mid C_n$ *uno nuevo. Se dice que* A *es* cerrado *si* $\bigcup C_i.\overline{\mathcal{R}} \subseteq \bigcup C_i.\mathcal{R}$.

En lo que sigue, se utiliza una notación de inclusión conjuntista entre conjuntos de interfaces para facilitar la legibilidad de las definiciones.

Definición 5 (Inclusión de interfaces) *Sean* $\mathcal{R}_1$ *y* $\mathcal{R}_2$ *dos conjuntos de interfaces con* $\mathcal{R}_1 = \{R_1^1, ..., R_1^s\}$ *y* $\mathcal{R}_2 = \{R_2^1, ..., R_2^t\}$*. Se dice que* $\mathcal{R}_1 \subseteq \mathcal{R}_2$ *si existe un* $\mathcal{I}$ *de* $\mathcal{R}_2$ *tal que para todo* $R \in \mathcal{R}_1$ *existe un* $R' \in \mathcal{I}$ *tal que* $R \sqsubseteq R'$.

Definición 6 (Intersección de interfaces) *Sean* $\mathcal{R}_1$ *y* $\mathcal{R}_2$ *dos conjuntos de interfaces,* $\mathcal{R}_1 = \{R_1^1, ..., R_1^s\}$ *y* $\mathcal{R}_2 = \{R_2^1, ..., R_2^t\}$*. Se define la intersección de interfaces* $\mathcal{R} = \mathcal{R}_1 \cap \mathcal{R}_2$ *como el conjunto* $\mathcal{R} = \{R^1, \cdots, R^u\}$ *tal que para todo* $R^i \in \mathcal{R}$ *existen dos interfaces* $R_1^j \in \mathcal{R}_1$ *y* $R_2^k \in \mathcal{R}_2$ *tales que* $R^i \equiv R_1^j$ *y* $R^i \equiv R_2^k$.

Con esto ya se puede definir el operador de reemplazabilidad extendido como:

Definición 7 (Reemplazabilidad de componentes) *Sean* $C_1 = (\mathcal{R}_1, \overline{\mathcal{R}}_1)$ *y* $C_2 = (\mathcal{R}_2, \overline{\mathcal{R}}_2)$ *dos componentes. Se dice que* C_1 *es* reemplazable *por* C_2*, y se denota por* $C_1 \sqsubseteq C_2$*, si* $(C_1.\mathcal{R}_1 \subseteq C_2.\mathcal{R}_2) \wedge (C_1.\overline{\mathcal{R}}_1 \supseteq C_2.\overline{\mathcal{R}}_2)$

3.3 Estrategias de composición con múltiples interfaces

Una vez establecido el ámbito de las interfaces de los componentes COTS y definido algunas de las operaciones sobre interfaces, que pueden utilizar los procesos *traders* en sus tareas de localización de componentes desde repositorios, en esta sección se van a estudiar las estrategias de composición.

Bajo el contexto descrito, una estrategia de composición debe resolver la forma de enfrentar las *especificaciones abstractas* de los componentes, que definen la arquitectura de la aplicación objeto ($\mathcal{A}$), y las *especificaciones concretas* de unos componentes que residen en los repositorios. Por simplicidad, se va a suponer que existe un único repositorio ($\mathcal{B}$). El objetivo es conseguir un subconjunto de componentes de $\mathcal{B}$ que ofrezcan los servicios definidos en la aplicación. A cada uno de estos subconjuntos se les va a denominar *configuración*. Según esto, dada una aplicación $\mathcal{A}$ definida por un conjunto de componentes 'abstractos' $\{A_1, A_2, \cdots, A_n\}$, el proceso de composición va a quedar definido por tres fases: selección de *candidatos*, generación de *configuraciones*, y cierre de las mismas.

Selección de componentes candidatos. En primer lugar, se debe seleccionar del repositorio $\mathcal{B}$ el conjunto de componentes $\{B_1, \cdots, B_m\}$ que puedan potencialmente formar parte de la aplicación $\mathcal{A}$ al ofrecer alguno de sus servicios. Este conjunto se denota por $C_{\mathcal{B}}(\mathcal{A})$. Para ello, se consideran solamente los servicios que ofrecen los componentes de $\mathcal{A}$, sin tener en cuenta los que requieren.

Considérese $\mathcal{A}$ como un componente $\mathcal{A} = A_1 \mid A_2 \mid \cdots \mid A_n$, donde $\mathcal{A}.\mathcal{R}$ y $\mathcal{A}.\overline{\mathcal{R}}$ representan los conjuntos de interfaces ofrecidos y los requeridos por la aplicación. Se define la *colección de componentes candidatos* de la aplicación $\mathcal{A}$ respecto al repositorio $\mathcal{B}$ como $C_{\mathcal{B}}(\mathcal{A}) = \{B \in \mathcal{B} \mid A.\mathcal{R} \cap B.\mathcal{R} \neq \varnothing\}$, esto es, los componentes del repositorio $\mathcal{B}$ que ofrecen algún servicio que también ofrece $\mathcal{A}$. Para construir este conjunto, se recorre el repositorio de componentes $\mathcal{B}$, decidiendo si se incluyen sus elementos en $C_{\mathcal{B}}(A)$. Expresado en términos de operaciones de *reemplazabilidad sintáctica*, la complejidad del proceso que construye la colección candidata es $O(m)$, siendo m el total de servicios disponibles en $\mathcal{B}$. En términos de *reemplazabilidad semántica* dicha complejidad es exponencial.

Generación de configuraciones. Las *configuraciones* son subconjuntos de componentes de entre los candidatos $C_{\mathcal{B}}$ que satisfacen las especificaciones de la aplicación completa y que no presentan solapamientos entre sus servicios (para componerlos sin problemas). Un posible algoritmo que calcula las configuraciones consiste en ir generando combinaciones entre los servicios de los componentes candidatos y escoger aquellas soluciones $Sol = \{C_1, \cdots, C_s\}$ con $A.\mathcal{R} = Sol.\mathcal{R}$.

```
1   function buscaConfiguraciones(i, Sol, S)
2     /* 1 ≤ i ≤ k, nivel en componentes. Sol es la configuración actual. */
3     if i ≤ k then
4       for j=1 to #C_i.R do // j es el nivel para los servicios del componente i
5         // Se trata de incluir el servicio C_i.R_j en la configuración.
6         if C_i.R_j ∉ Sol.R then // ¿Está el servicio R_j en R?
7           Sol := Sol ∪ {C_i.R_j}
8           if A.R ⊆ Sol.R then // si la configuración obtenida ya es válida ...
9             S := S ∪ {Sol} // se incluye en el conjunto de configuraciones
10          else // si no se han cubierto todos los servicios de la aplicación A...
11            buscaConfiguraciones(i, Sol, S) // sigue buscando en C_i ...
12          endif;
13          Sol := Sol − {C_i.R_j}
14        endif;
15      endfor // sigue buscando en el componente C_i.
16      buscaConfiguraciones(i + 1, Sol, S) // Siguiente componente.
17    endif
18  endfunction
```

Figura1. Algoritmo para la generación de configuraciones.

El algoritmo de *backtracking* que se muestra en la figura 1, genera el conjunto $\mathcal{S}$ de todas las configuraciones posibles a partir de la aplicación $\mathcal{A}$ —considerada como un componente— y de los componentes que hay en la colección de componentes candidatos $C_{\mathcal{B}}(A) = \{C_1, \cdots, C_k\}$. La llamada inicial del algoritmo es: $Sol = \varnothing$; $\mathcal{S} = \varnothing$; `buscaConfiguraciones`$(1, Sol, \mathcal{S})$.

Como se puede ver, el algoritmo explora todas las posibilidades y construye una lista de configuraciones válidas (línea 9). Cada configuración individual (línea 7) se genera al recorrer todos los servicios de los componentes de la colección candidata, incorporando en la solución parcial aquellos servicios del componente que no están en la misma —en la forma $C_i.R_j$— y descartando aquellos que ya están (líneas 6 y 8). Al finalizar, $\mathcal{S}$ contiene todas las configuraciones válidas. Luego, para cada una de ellas, es necesario ocultar los servicios de componente que ya están presentes en la misma. Una implementación de estos procesos se encuentra disponible en `http://www.ualm.es/~liribarn`.

Por la forma en la que se ha presentado el proceso, no se producen solapamientos entre servicios ni se obtienen configuraciones 'parciales', sólo las que ofrecen los servicios de A. Respecto al orden de complejidad del algoritmo, expresado en términos de operaciones de reemplazabilidad, su tiempo de ejecución en el peor de los casos, es de orden $O(2^n)$, siendo n el número total de servicios del conjunto de componentes candidatos $C_{\mathcal{B}}(A)$.

Cierre de las configuraciones. Una vez obtenidas las configuraciones, es preciso cerrarlas para formar una aplicación completa, y evitar la existencia de lagunas en el caso de no estar presentes los servicios requeridos por algún componente de la configuración concreta. Para ello es suficiente utilizar cualquiera de los algoritmos existentes para el cierre transitivo de un conjunto con respecto a otro, en este caso con respecto a los componentes del repositorio $\mathcal{B}$.

3.4 Algunas consideraciones sobre las configuraciones.

Una vez establecido un proceso para extraer un conjunto de configuraciones a partir de la especificación abstracta de una aplicación desde un repositorio de componentes, en esta sección se discute algunos aspectos de las mismas.

Métricas sobre configuraciones. En general, el proceso que aquí se propone ofrece al diseñador de aplicaciones un conjunto de configuraciones posibles desde donde poder seleccionar aquella que se ajuste mejor a sus necesidades. Sin embargo, y puesto que el algoritmo que las construye es exponencial, sería muy interesante disponer de métricas capaces de asignar *pesos* a cada una de las configuraciones. Por ejemplo, se podrían considerar factores como el precio de cada componente, su complejidad (p.e. en función del número de interfaces que soportan), su fabricante, etc. Esto permitiría, entre otras cosas, presentar las soluciones ordenadas según un criterio definido por el usuario, y si se dispone de pesos, también se podría modificar el algoritmo para extenderlo a uno de ramificación y poda, para eliminar aquellas opciones que no conduzcan a configuraciones deseables, y rebajar también la complejidad del algoritmo.

Adecuación a la estructura interna de la aplicación Las configuraciones se han construido a partir de los servicios que ofrece la aplicación, sin tenerse en cuenta para nada su estructura interna. Cabría preguntarse también si es posible generar configuraciones que respeten esa estructura, en el sentido que se mantenga la *división* en los componentes que se especifican en la definición $A = \{A_1, \cdots, A_n\}$. Para ello, se pueden generar las configuraciones siguiendo el proceso anterior, y descartar aquellas que no sean *compatibles* con la arquitectura definida para A. El concepto de compatibilidad en este contexto se define como:

Definición 8 *Sea* $A = \{A_1, \cdots, A_n\}$ *una aplicación y* $S = \{S_1, \cdots, S_m\}$ *una configuración para* A*, esto es, un conjunto de componentes del repositorio cuya composición ofrece los mismos servicios que* A*, y que no presenta solapamientos entre ellos. Se dice que* S *respeta la estructura de* A *si* $\forall\, i \in \{1..m\}, \forall\, j \in \{1..n\} \bullet S_i.\mathcal{R} \cap A_j.\mathcal{R} \neq \varnothing \Rightarrow (S_i \sqsubseteq A_j) \vee (A_j \sqsubseteq S_i)$

Esta definición obliga a que los componentes de la configuración estén 'contenidos' o 'contengan' a las especificaciones abstractas de los componentes de la aplicación, respetando las 'fronteras' definidas en la arquitectura de la aplicación.

4 Un ejemplo

En este apartado se describe un ejemplo que ilustra los conceptos definidos. El ejemplo se centra en la especificación de un escenario personalizado al estilo de un Escritorio. Para ello, supóngase que se desea un Escritorio con 4 componentes:

$E = \{$*Calculadora (CAL), Calendario (CIO), Agenda (AG), Meeting Scheduler (MS)*$\}$

En la figura 2 se ilustra un esquema de las interdependencias para los cuatro componentes del ejemplo propuesto. En ella, cada componente de la arquitectura impone sus requisitos de servicios externos, expresados por un círculo blanco (esto es, los servicios externos que necesita para funcionar) mientras que los servicios que ofrece cada componente se expresan con un círculo en negro. También se indican con flechas los enlaces entre unos y otros.

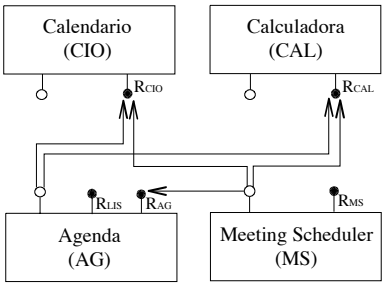

Figura2. Arquitectura del caso ejemplo.

De aquí se desprenden las restricciones para los 4 componentes abstractos de la arquitectura: el *Calendario* (CIO) implementa el servicio R_{CIO} y no requiere ningún otro componente para funcionar; la *Calculadora* (CAL) implementa el servicio R_{CAL} y tampoco requiere otro componente; la *Agenda* (AG) implementa R_{AG} y R_{LIS} –una agenda y un listín telefónico– y necesita una calculadora R_{CAL} y un calendario R_{CIO}; por último, el *Meeting Scheduler* (MS) implementa R_{MS} y necesita una agenda R_{AG}, una calculadora R_{CAL} y un calendario R_{CIO}.

Una vez descrito el ejemplo, se procede según las fases anteriores: (1) selección de los componentes candidatos; (2) generación de las configuraciones; y (3) cierre de las configuraciones. En la primera fase, la aplicación se considera como un componente. Para ello, se efectúa la composición de todos los componentes del escritorio E, obteniendo el componente $E.\mathcal{R} = \{R_{CIO}, R_{CAL}, R_{AG}, R_{LIS}, R_{MS}\}$, $E.\overline{\mathcal{R}} = \{\}$. Luego, se enfrentan estos servicios con los que ofrecen los componentes de $\mathcal{B}$. El resultado de este proceso se muestra en la tabla 1. A la izquierda, se incluyen las *especificaciones abstractas* de los componentes que definen la arquitectura ejemplo, correspondientes a la figura 2. La columna de la derecha contiene un posible resultado de la fase de *selección de candidatos* desde el repositorio $\mathcal{B}$, con 6 *especificaciones concretas* de componente.

Arquitectura (especificaciones abstractas)	$C_{\mathcal{B}}(E)$: Colección de candidatos (especificaciones concretas)
$CIO = \{R_{CIO}\}$ $CAL = \{R_{CAL}\}$ $AG = \{R_{AG}, R_{LIS}, \overline{R}_{CAL}, \overline{R}_{CIO}\}$ $MS = \{R_{MS}, \overline{R}_{AG}, \overline{R}_{CAL}, \overline{R}_{CIO}\}$	$C_1 = \{R_{CIO}\}$ $C_2 = \{R_{CAL}\}$ $C_3 = \{R_{AG}, R_{CIO}, \overline{R}_{CAL}\}$ $C_4 = \{R_{LIS}\}$ $C_5 = \{R_{MS}, R_{AG}, \overline{R}_{CIO}\}$ $C_6 = \{R_{CAL}, R_{LIS}, \overline{R}_P\}$

Tabla1. Las especificaciones de los componentes de la arquitectura y de los candidatos.

Obsérvese que en $C_{\mathcal{B}}$ se ha seleccionado un componente (C_6) que requiere el servicio R_P para que pueda funcionar. Esto quiere decir que si en la siguiente fase (*generación de configuraciones*) se detecta una configuración que la incluya, se originará una *laguna*, pues no existe ningún componente en $C_{\mathcal{B}}$ que pueda ofrecer el servicio requerido (ninguno ofrece R_P). No obstante, este problema se resuelve en la última fase, con el *cierre* de la configuración respecto al repositorio.

Para el ejemplo propuesto, en la tabla 2 se ilustra una parte del resultado que genera el algoritmo `buscaConfiguraciones`. La primera columna de la tabla es un número para las configuraciones (en cursiva las que dan lugar a configuraciones válidas). Las columnas 2 a 7 indican el servicio que aporta cada componente candidato. Así por ejemplo, en la configuración 5 intervienen todos los componentes de $C_{\mathcal{B}}$, aportando cada uno de ellos un servicio, excepto C_6, cuyos servicios ya han sido cubiertos en la configuración resultante. En la tabla, los componentes que intervienen en las configuraciones han sido incluidos directamente con los servicios correspondientes ocultos (última columna).

Este algoritmo genera configuraciones completas, ya que trata de cubrir todos los servicios de la aplicación. En caso contrario, descarta la configuración, como ocurre en las combinaciones 4, 33 y 80, entre otras. Para este ejemplo se obtienen finalmente 16 configuraciones válidas distintas, de las cuales 4 son cerradas (5, 11, 75 y 93) y el resto no (las que incorporan el componente C_6, por requerir el servicio R_P). Además, hay 4 configuraciones que respetan la arquitectura (5, 7, 16 y 26). A partir de aquí, es ya decisión del diseñador seleccionar aquella configuración que más le interese utilizar para construir su aplicación.

	C_1	C_2	C_3	C_4	C_5	C_6	Configuraciones $\mathcal{S}$
4	R_{CIO}	R_{CAL}	R_{AG}	R_{LIS}	-	-	$\varnothing$ – Falta R_{MS} (laguna)
5	R_{CIO}	R_{CAL}	R_{AG}	R_{LIS}	R_{MS}	-	C_1, C_2, C_3–$\{R_{CIO}\}, C_4, C_5$–$\{R_{AG}\}$
7	R_{CIO}	R_{CAL}	R_{AG}	-	R_{MS}	R_{LIS}	C_1, C_2, C_3–$\{R_{CIO}\}, C_5$–$\{R_{AG}\}, C_6$–$\{R_{CAL}\}$
11	R_{CIO}	R_{CAL}	-	R_{LIS}	R_{MS}, R_{AG}	-	$C_1, C_2, C_4, C5$
16	R_{CIO}	R_{CAL}	-	-	R_{MS}, R_{AG}	R_{LIS}	C_1, C_2, C_5, C_6–$\{R_{CAL}\}$
26	R_{CIO}	-	R_{AG}	R_{LIS}	R_{MS}	R_{CAL}	C_1, C_3–$\{R_{CIO}\}, C4, C_5$–$\{R_{AG}\}, C_6$–$\{R_{LIS}\}$
30	R_{CIO}	-	R_{AG}	-	R_{MS}	R_{CAL}, R_{LIS}	C_1, C_3–$\{R_{CIO}\}, C_5$–$\{R_{AG}\}, C_6$
33	R_{CIO}	-	R_{AG}	-	-	R_{LIS}	$\varnothing$ – Faltan R_{CAL}, R_{MS} (laguna)
40	R_{CIO}	-	-	R_{LIS}	R_{MS}, R_{AG}	R_{CAL}	C_1, C_4, C_5, C_6–$\{R_{LIS}\}$
50	R_{CIO}	-	-	-	R_{MS}, R_{AG}	R_{CAL}, R_{LIS}	C_1, C_5, C_6
75	-	R_{CAL}	R_{AG}, R_{CIO}	R_{LIS}	R_{MS}	-	C_2, C_3, C_4, C_5–$\{R_{AG}\}$
77	-	R_{CAL}	R_{AG}, R_{CIO}	-	R_{MS}	R_{LIS}	C_2, C_3, C_5–$\{R_{AG}\}, C_6$–$\{R_{CAL}\}$
80	-	R_{CAL}	R_{AG}	R_{LIS}	R_{MS}	-	$\varnothing$ – Falta R_{CIO} (laguna)
93	-	R_{CAL}	R_{CIO}	R_{LIS}	R_{MS}, R_{AG}	-	C_2, C_3–$\{R_{AG}\}, C_4, C_5$
98	-	R_{CAL}	R_{CIO}	-	R_{MS}, R_{AG}	R_{LIS}	C_2, C_3–$\{R_{AG}\}, C_5, C_6$–$\{R_{CAL}\}$
123	-	-	R_{AG}, R_{CIO}	R_{LIS}	R_{MS}	R_{CAL}	C_3, C_4, C_5–$\{R_{AG}\}, C_6$–$\{R_{LIS}\}$
127	-	-	R_{AG}, R_{CIO}	-	R_{MS}	R_{CAL}, R_{LIS}	C_3, C_5–$\{R_{AG}\}, C_6$
165	-	-	R_{CIO}	R_{LIS}	R_{MS}, R_{AG}	R_{CAL}	C_3–$\{R_{AG}\}, C_4, C_5, C_6$–$\{R_{LIS}\}$
175	-	-	R_{CIO}	-	R_{MS}, R_{AG}	R_{CAL}, R_{LIS}	C_3–$\{R_{AG}\}, C_5, C_6$
210	-	-	-	-	-	R_{CAL}, R_{LIS}	$\varnothing$ – Faltan R_{CIO}, R_{AG}, R_{MS} (laguna)

Tabla2. Resultado del algoritmo `buscaConfiguraciones()` para el ejemplo.

El proceso aquí definido ha sido mostrado para trabajar con aplicaciones completas. Sin embargo, también es posible utilizarlo sobre ciertas partes de una aplicación. De esta forma, se deja libertad al diseñador para poder decidir, a partir de una arquitectura dada, qué partes de la aplicación desea implementar mediante componentes del repositorio, y cuáles no.

5 Trabajos relacionados

Las contribuciones que se han presentado en este artículo pueden relacionarse con dos líneas de investigación. En primer lugar, se encuentran aquellos trabajos que tratan con la reemplazabilidad de componentes a cualquier nivel, discutidos en la sección 2. Este trabajo trata de extender dichas propuestas para el caso de que los componentes ofrezcan (y requieran) múltiples interfaces.

En segundo lugar, se encuentran los trabajos que tratan con la búsqueda y selección de componentes en repositorios software, como por ejemplo en [15], donde se propone una técnica que permite razonar semánticamente durante los procesos de selección y ensamblaje sobre aquellos componentes que cumplen los requisitos del sistema. Estos requisitos son recogidos mediante unos diagramas

de transiciones llamados 'mapas' que se basan en cuatro modelos: el modelo As-Is, el modelo To-Be, el modelo COTS y el modelo match integrado. Sus inconvenientes: (a) es una propuesta muy abstracta; (b) no hace una propuesta de especificación COTS; y (c) no describe detalles de cómo se llevan a cabo los emparejamientos sintácticos y semánticos para comprobar la reemplazabilidad e interoperabilidad de componentes, entre otros. También destaca el trabajo de Goguen [7], en donde se establecen criterios de selección para la búsqueda de componentes en repositorios. Una vez más, dichos criterios se basan en interfaces simples, y por tanto no contemplan los problemas de solapamientos o lagunas.

6 Conclusiones y trabajo futuro

El trabajo aquí presentado pretende establecer las bases desde donde acometer la definición de una *metodología* para el desarrollo de aplicaciones software basado en COTS, a partir de la *especificación* de su arquitectura software. Este trabajo presenta dos contribuciones principales. En primer lugar, se han extendido los tradicionales operadores de reemplazabilidad y equivalencia de componentes para el caso de que los componentes ofrezcan y requieran *múltiples interfaces*, complementando los estudios tradicionales en los que los componentes sólo ofrecían una interfaz y no se tenían en cuenta los servicios que necesitan de otros componentes para funcionar. Como se ha mostrado, aparecen nuevos problemas, como los *solapamientos* o las *lagunas* de las interfaces. Asimismo, se ha presentado un algoritmo para la *generación de configuraciones* a partir de las *especificaciones abstractas* de los componentes de una aplicación –según se definen en su arquitectura software– y de las *especificaciones concretas* de los componentes que están registrados en un repositorio.

La presente contribución pretende ser extendida en varias direcciones. En primer lugar, para trabajar con *especificaciones formales* de las arquitecturas software, como son las que ofrecen los lenguajes de descripción de arquitecturas (LDAs): por ejemplo Rapide, Darwin o LEDA. En segundo lugar, se pretende extender las especificaciones de los componentes, de forma que puedan contener información no sólo a nivel de *signaturas*, sino también a nivel de *protocolos* o *semántico*. En este sentido, se pretende trabajar en la búsqueda de *plantillas de especificación* de componentes, en la línea de Dong [6]. Basado en este tipo de especificaciones, es necesario extender los *repositorios* actuales para que puedan almacenar y manejar ese tipo de información, así como construir *traders* que permitan buscar servicios a partir de este tipo de especificaciones. Y por último, es necesario definir métricas y heurísticas para poder comparar *configuraciones*, proporcionando al diseñador de aplicaciones mejores herramientas para la toma de decisiones sobre los componentes que ha de incorporar en la aplicación.

Agradecimientos. Nuestro agradecimiento a los revisores del JIS2000 por sus comentarios y aclaraciones sobre el articulo que han sido de gran utilidad para mejorar la presente contribución. Este trabajo ha sido subvencionado en parte por el proyecto de investigación CICYT TIC99-1083-C02-01.

Referencias

1. P. America. Designing an object-oriented programming language with behavioral subtyping. In *Foundations of Object-Oriented Languages*, pages 60–90. LNCS 489, Springer Verlag, 1991.
2. R. Anaya. *Desarrollo y gestión de componentes reutilizables en el marco de OASIS*. PhD thesis, Universidad Politécnica de Valencia, 1999.
3. R. Bastide, O. Sy, and P. Palanque. Formal specification and prototyping of CORBA systems. In *Proceedings of ECOOP'99*, number 1628 in LNCS, pages 474–494. Springer-Verlag, 1999.
4. C. Canal, L. Fuentes, J. M. Troya, and A. Vallecillo. Extending CORBA interfaces with π-calculus for protocol compatibility. In *Proc. of TOOLS'00*, France, June 2000. IEEE Computer Society Press.
5. K. K. Dhara and G. T. Leavens. Forcing behavioral subtyping through specification inheritance. In *Proceedings of the 18th International Conference on Software Engineering (ICSE-18)*, pages 258–267, Berlin, Germany, 1996. IEEE Press.
6. J. Dong, P. Alencar, and D. Cowan. A component specification template for COTS-based software development. In *Proc. of the Ensuring Successful COTS Development Workshop*, 1999.
7. J. Goguen, D.Nguyen, J. Meseguer, Luqi, D. Zhang, and V. Berzins. Software component search. *Journal of Systems integration*, 6:93–134, September 1996.
8. J. Han. Semantic and usage packaging for software components. In A. Vallecillo, J. Hernández, and J. M. Troya, editors, *Proc. of the ECOOP'99 Workshop on Object Interoperability (WOI'99)*, pages 25–34, June 1999.
9. D. Lea. Interface-based protocol specification of open systems using PSL. In *Proc. of ECOOP'95*, number 1241 in LNCS. Springer-Verlag, 1995.
10. E. Manso, F. J. García, M. P. Romay, and J. M. Marqués. Modelo de cualifiación de auditorías de elementos reutilizables de un repositorio. In *IV Jornadas de Ingenería del Software y Bases de Datos (JISBD'99)*, 1999.
11. A. Mikhajlova. *Ensuring Correctness of Object and Component Systems*. PhD thesis, Åbo Akademi University, October 1999.
12. H. Mili, F. Mili, and A. Mili. Reusing software: Issues and research directions. *IEEE Trans. on Software engineering*, 21(6):528–562, June 1995.
13. OMG. *The CORBA Component Model*, June 1999. http://www.omg.org.
14. S. Robertson and J. Robertson. *Mastering the Requirement Process*. Addison-Wesley, 1999.
15. C. Rolland. Requirements engineering for COTS based systems. *Information and Software Technology. Elsevier.*, 41:985–990, 1999.
16. C. Szyperski. *Component Software*. Addison-Wesley, 1998.
17. A. Vallecillo, J. Hernández, and J. M. Troya, editors. *Proc. of the ECOOP'99 Workshop on Object Interoperability*, June 1999.
18. K. C. Wallnau, D. Carney, and B. Pollack. How COTS software affects the design of COTS-intensive systems. SEI Interactive, 1998.
19. D. M. Yellin and R. E. Strom. Protocol specifications and components adaptors. *ACM Trans. on Programming Languages and Systems*, 19(2):292–333, March 1997.
20. A. M. Zaremski and J. M. Wing. Specification matching of software components. *ACM Trans. on Software Engineering and Methodology*, 6(4):333–369, Oct. 1997.