Mediación semántica A* basada en MDE para la generación de arquitecturas en tiempo de ejecución

Javier Criado, Luis Iribarne, Nicolás Padilla

Grupo de Informática Aplicada, Universidad de Almería, España
{javi.criado,luis.iribarne,npadilla}@ual.es

Resumen. Existen aplicaciones software que necesitan ser adaptadas en tiempo de ejecución debido a que los componentes que conforma su arquitectura no representan la configuración óptima. En estos casos, las arquitecturas deben ser reconfiguradas, por ejemplo, añadiendo y eliminando componentes, o modificando las relaciones entre ellos. Este artículo presenta una propuesta para la generación de arquitecturas en tiempo de ejecución. Está enfocado en la descripción del proceso que ocurre desde que existe una definición de arquitectura que hay que resolver, hasta que se genera la mejor configuración que da solución a dicha arquitectura. Para construir dicha configuración, se utilizan técnicas de modelado, mecanismos de *trading* y un algoritmo de búsqueda A*. Dicho algoritmo hace uso de una heurística basada en la información sintáctica y semántica de los componentes. Como dominio de aplicación, se muestra un caso estudio para la generación de interfaces de usuario.

Palabras clave: Componentes, Trading, Heurística, A*, MDE, M2M

1 Introducción

El Desarrollo de Software Basado en Componentes (DSBC) permite construir aplicaciones a partir de la unión y conexión de partes más pequeñas [9]. Algunos de estos sistemas requieren que la gestión de los componentes que los constituyen ocurra de forma dinámica [10]. Esto sucede, por ejemplo, cuando las aplicaciones no sólo se construyen en tiempo de diseño (por parte de un desarrollador de software), sino que también deben adaptarse en tiempo de ejecución a los cambios producidos en el contexto. Para estos casos, cada vez que se necesita adaptar la arquitectura, deben seleccionarse los componentes idóneos a partir de un conjunto de componentes disponibles.

Existen, por tanto, uno o varios repositorios de componentes a los que el sistema accede para determinar cuál es la mejor configuración de componentes posible. Dichos repositorios no tienen que permanecer estáticos, sino que los elementos que los conforman pueden variar en el tiempo. Por este motivo, el proceso de selección mencionado no siempre generará la misma solución ante una misma entrada, sino que el resultado dependerá (entre otros factores) de los componentes que existan en los repositorios.

Puesto que la adaptación se realiza en tiempo de ejecución, la construcción de nuevas configuraciones de componentes debe ser un proceso limitado en el tiempo y que garantice la generación de una solución válida [21]. Este artículo presenta una propuesta para la generación de configuraciones de arquitecturas software en tiempo de ejecución. Se propone un proceso de mediación semántica que extiende el modelo de mediación (*trading*) [22] tradicional incluyendo un algoritmo A* para el cálculo de configuraciones. Este proceso nos permite gestionar las especificaciones de los componentes a través de su consulta, registro y administración. Por otro lado, el algoritmo A* es utilizado como parte de la funcionalidad que construye las posibles configuraciones de salida y que comprueba el grado de similitud entre la arquitectura buscada y la solución generada (ver Figura 1). Para ello, el algoritmo hace uso de una función heurística que calcula la distancia con respecto a la arquitectura de referencia. Dicha heurística tiene en cuenta información sintáctica y semántica de los componentes.

A este proceso se le ha llamado *regeneración* y se enmarca dentro de una propuesta de metodología para la adaptación en tiempo de ejecución de arquitecturas de componentes. Dicha adaptación está compuesta por dos fases principales: una primera fase de *transformación* que se encarga de adaptar las definiciones abstractas de las arquitecturas software (por ejemplo, insertando nuevos componentes, eliminando otros o cambiando las conexiones entre ellos), y una segunda fase de *regeneración* que resuelve las arquitecturas concretas a partir de las definiciones abstractas (ver Figura 2). Este artículo se centra en la descripción de la fase de ***regeneración***, partiendo de una arquitectura abstracta que necesita ser resuelta en una arquitectura concreta. La metodología, en su conjunto, está pensada para arquitecturas de componentes en general, pero en este artículo nos vamos a centrar en el dominio de aplicación de las interfaces de usuario basadas en componentes tipo *mashups* [11]. Aunque la Figura 2 muestra

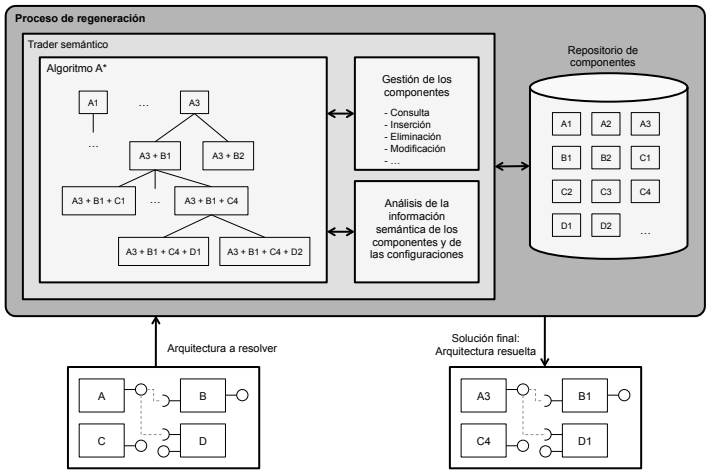

Fig. 1. Mediación semántica A* para la generación de arquitecturas software

Fig. 2. Metodología para la adaptación de arquitecturas en tiempo de ejecución

una visualización de la arquitectura concreta, no es objetivo de este artículo la descripción del proceso final del despliegue de la arquitectura [34]. Por otro lado, en [8, 30] puede encontrarse más información acerca de la fase de transformación.

Toda la metodología ha sido desarrollada haciendo uso de técnicas de *Model-Driven Engineering* (MDE). Esta elección se debe a que, desde el principio, nuestra propuesta está orientada a tratar de facilitar el diseño, la construcción, el despliegue y la reutilización de las arquitecturas software. En este sentido, estas técnicas permiten describir la estructura de las arquitecturas a través de sus componentes, sus interfaces y sus relaciones, o definir las características propias de cada componente, por ejemplo, en base a sus propiedades funcionales y extra-funcionales. Por este motivo, todos los elementos que maneja el proceso de *regeneración* (*i.e.*, especificaciones de componentes, arquitecturas abstractas y concretas, etc.) han sido definidos a través de modelos y son manipulados (creados, transformados, validados, etc.) gracias al uso de técnicas MDE.

El resto del artículo se organiza de la siguiente forma. La Sección 2 define las bases de nuestra propuesta. La Sección 3 describe el proceso de *trading* desarrollado para la resolución de arquitecturas en tiempo de ejecución. La Sección 4 presenta un caso de estudio para la aplicación de la propuesta en interfaces de usuario basadas en componentes. La Sección 5 revisa el trabajo relacionado y, por último, la Sección 6 resume las conclusiones y enumera algunas opciones posibles de trabajo futuro.

2 Bases de la metodología

En esta sección se definen cuáles son los principios que deben existir para que sea posible la aplicación de nuestra metodología. Además, se presentan los elementos de modelado principales que participan en la fase de regeneración.

2.1 Pre-condiciones para la aplicación de la metodología

La metodología mencionada anteriormente ha sido desarrollada con un enfoque generalista para la adaptación de sistemas software basados en componentes. Por tanto, el proceso de adaptación puede ser utilizado en diferentes dominios. No obstante, es necesario que existan una serie de pre-condiciones para que la metodología pueda ser aplicada:

(a) La aplicación debe poder ser representada únicamente a través componentes y sus relaciones, *i.e.*, a través de una arquitectura software.
(b) Las relaciones entre componentes determinan la validez de una determinada arquitectura. Si un componente necesita ser obligatoriamente vinculado con otro componente que no está presente, la arquitectura no será válida.
(c) El conjunto de componentes que dan solución a una arquitectura varía dependiendo de las relaciones entre los componentes de la arquitectura y de los componentes disponibles.
(d) La comunicación entre componentes se realiza a través de la invocación de las operaciones descritas en sus interfaces.
(e) Los componentes deben describirse a partir de una especificación basada en propiedades funcionales y extra-funcionales.
(f) Cualquier desarrollador podrá implementar componentes de este tipo publicando únicamente la información relativa al componente, de forma similar al uso de componentes COTS (Commercial Off-The-Shelf) [2].
(g) Los componentes deben poder ser almacenados en un repositorio y utilizados en tiempo de ejecución para la construcción de las arquitecturas.
(h) Las plataformas donde se despliegan los componentes, deben permitir la modificación de las arquitecturas en tiempo de ejecución.

Algunos posibles dominios de aplicación son los entornos domóticos [4], aplicaciones de *smart* TV [19], ciudades inteligentes [13], redes de comunicación [15], o interfaces de usuario [18]. No obstante, tanto el proceso de regeneración que se presenta en este artículo como la metodología mencionada han sido aplicados hasta el momento al dominio de la interfaces gráficas de usuario (GUI) basadas en componentes. Entiéndase que la metodología no puede aplicarse en cualquier tipo de GUIs, sino únicamente a aquellas que cumplan con los requisitos mencionados anteriormente. Un ejemplo de este tipo de GUI es un Sistema de Información Geográfica (SIG) en el que existen componentes de tipo mapa, histogramas, visualizadores de leyendas, componentes de comunicación, generadores de informes, etc., como la aplicación desarrollada para el proyecto de investigación ENIA (`http://acg.ual.es/enia/gui`). En este dominio existen dependencias entre componentes, por ejemplo el componente leyenda sólo puede existir en la arquitectura si tiene un componente mapa asociado (puesto que se muestra la leyenda de las capas visualizadas en el mapa). Por otro lado, dependiendo de los mapas disponibles y de la arquitectura que quiera construirse, la selección del mejor mapa puede variar.

2.2 Principales modelos que participan en el proceso

En el proceso de regeneración intervienen cuatro tipos de modelos para la representación de arquitecturas abstractas, arquitecturas concretas, componentes abstractos y componentes concretos. El término "abstracto" identifica lo que queremos que esté presente en la arquitectura, mientras que "concreto" define a aquellos objetos reales que forman parte de una solución arquitectónica. Por limitaciones de espacio, las imágenes que representan los metamodelos para las arquitecturas y para los componentes, se encuentran en `http://acg.ual.es/jisbd2015`. Los modelos de arquitectura abstracta (AAM) describen qué componentes se encuentran presentes en la arquitectura y cómo se relacionan entre ellos a través de dependencias entre sus interfaces. Las interfaces proporcionadas definen qué servicios ofrece el componente, mientras que las requeridas representan los servicios a los que el componente debe poder acceder para funcionar de forma correcta. Los modelos de arquitectura concreta (CAM), además de la información anterior, también describen las relaciones entre componentes en término de puertos y conectores entre ellos.

Es necesario aclarar en este punto la pre-condición de nuestra metodología que se refiere a la comunicación entre componentes. Cuando nos referimos a que se realiza a través de la invocación de las operaciones descritas en sus interfaces, proponemos la siguiente solución para las arquitecturas de software concretas (que posteriormente serán desplegadas en la plataforma escogida). Las operaciones de cada componente son accesibles a través de un puerto. Las operaciones que no devuelven información como salida, se representa por un único puerto de entrada, mientras que las operaciones con entrada y salida, son representados por un puerto de entrada y otro de salida. Para las operaciones que pertenecen a una interfaz requerida, la representación es al contrario: un único puerto de salida para el primer caso e inversión del orden en el segundo caso. La Figura 3 muestra la correspondencia entre un AAM y un CAM, según el criterio establecido.

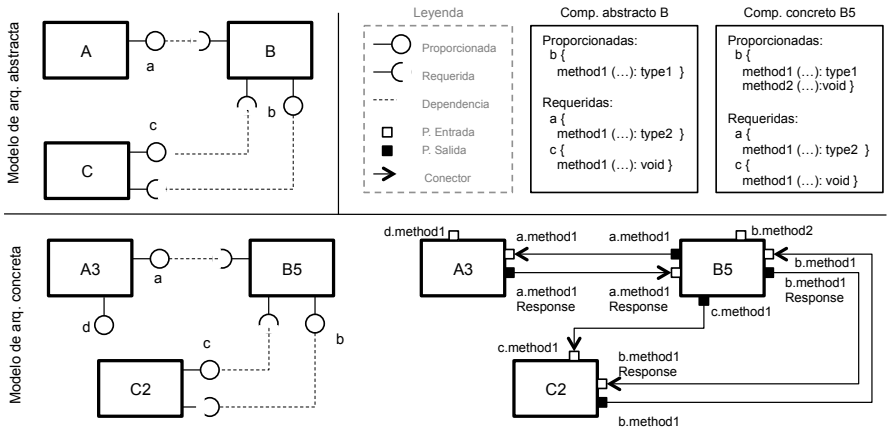

Fig. 3. Modelo de arquitectura abstracta y modelo de arquitectura concreto

Respecto a los componentes, una especificación de **componente abstracto** contiene el conjunto características que debe poseer un componente software, mientras que una especificación de **componente concreto** describe las características que posee un componente real ya implementado. Ambos conceptos tratan de equipararse, respectivamente, a las definiciones de *tipo de servicio* y de *servicio* proporcionadas por la especificación ODP [23]. El metamodelo que define las especificaciones de ambos tipos de componentes, está inspirado en modelos de documentación existentes para la descripción de servicios web [16] y en el modelo de documentación para componentes COTS propuesto en [22].

En nuestra propuesta, tanto los componentes abstractos como los concretos tienen cuatro partes principales. La primera describe características funcionales en términos de interfaces proporcionadas y requeridas. Cada interfaz está definida mediante el uso de WSDL (*Web Services Description Language*) [16]. La segunda parte contiene características extra-funcionales (no-funcionales, de calidad de servicio, etc.) del componente. Las propiedades definen atributos y sus correspondientes valores que los componentes tienen o deben tener. Las dependencias determinan cuáles de las interfaces requeridas son estrictamente obligatorias y deben ser resueltas en una arquitectura para que funcione correctamente. En nuestra propuesta, los componentes funcionan correctamente (aunque no de forma completa) si alguna de sus interfaces requeridas, que no se encuentra en la lista de dependencias, no ha sido resuelta en la arquitectura concreta.

La tercera parte define información de *packaging* incluyendo datos acerca de su implementación, su localización y su versión. Por último, la parte de *marketing* contiene información sobre la entidad desarrolladora del componente e información de sus contactos. A la hora de definir ambos tipos de componentes, existen algunas diferencias en la cardinalidad y obligatoriedad de algunas de las partes y de algunos de los atributos. Por ejemplo, es obligatorio definir los bloques de *packaging* y de *marketing* de los componentes concretos, mientras que en los abstractos ambas partes son opcionales. Dichos tipos de restricciones sintánticas son comprobadas y validadas gracias al uso de OCL (*Object Constraint Language*) [1].

3 Mediación semántica A* en tiempo de ejecución

Esta sección describe la fase de regeneración. Primero, se presenta el proceso de *trading* desarrollado; en segundo lugar, se muestra el uso de la información semántica dentro de este proceso de mediación. Por último, se describe el algoritmo A* implementado para la construcción de configuraciones de componentes.

3.1 Mediación en tiempo de ejecución

Los procesos de mediación o *trading* son mecanismos conocidos para la búsqueda y localización de servicios [27, 20]. En lo relativo a componentes, y en especial a componentes de terceros (tipo COTS), existen trabajos en la literatura que

abordan el uso de estos mecanismos para la gestión de componentes y el asesoramiento en la construcción de arquitecturas software en tiempo de diseño [22, 5]. El proceso de *trading* implementando en nuestra propuesta debe dar soporte a las tareas mencionadas (gestión de componentes y de arquitecturas), pero en tiempo de ejecución. Por este motivo, debe lidiar adicionalmente con algunos aspectos inherentes al tiempo. A continuación se describen las características principales de dicho proceso.

Una parte esencial de la gestión de componentes y de arquitecturas está fundamentado en la correcta manipulación de las especificaciones que describen cada uno de los componentes software que intervienen en el proceso. Para ello, el objeto software encargado de la mediación (*trader*) debe poder: realizar búsquedas de componentes a partir de parámetros de entrada, añadir/modificar/eliminar especificaciones de componentes, disponer de un mecanismo para configurar sus políticas de ejecución. Dichos bloques de funcionalidad se corresponden con la implementación de la funcionalidad indicada por las interfaces *Lookup*, *Register* y *Admin* del estándar de mediación ODP (Open Distributed Processing) [23].

Cuando un trader implementa las tres interfaces mencionadas, se habla de un servicio de mediación *standalone*. Nuestro *trader*, es un objeto de mediación de este tipo. Adicionalmente, para la obtención de configuraciones de arquitecturas válidas en tiempo de ejecución, el *trader* extiende el modelo de mediación tradicional incorporando una nueva interfaz *Configs*. Este módulo incluye toda la funcionalidad necesaria para la resolución de arquitecturas concretas a partir de definciones de arquitecturas abstractas y junto con la información ofrecida por las especificaciones de componentes abstractos y concretos. Está inspirado en el proceso de cálculo de configuraciones propuesto en [22], pero orientado a la generación de arquitecturas en tiempo de ejecución. La subsección 3.3 ofrece más detalles sobre su funcionamiento.

A continuación se listan las operaciones principales ofrecidas por el *trader*: i) Obtener el conjunto de especificaciones de componentes que cumplen con unos determinados valores según su metamodelo (*Lookup*); ii) Añadir/Modificar/Eliminar especificaciones de componentes (*Register*); iii) Obtener en tiempo de ejecución configuraciones de arquitecturas concretas a partir de arquitecturas abstractas (*Configs*); iv) Modificar el tamaño del conjunto de especificaciones devuelto por las operaciones de *Lookup* (*Admin*); v) Limitar el tiempo máximo en la construcción de la arquitectura concreta (*Admin*); vi) Modificar las características del *matching* que se realiza al construir las configuraciones de arquitecturas concretas (*Admin*).

3.2 Mediación semántica

Los modelos de documentación de componentes COTS describen las interfaces haciendo uso de caracterísitcas sintácticas y semánticas [3]. Por lo general, la información sintáctica se refiere a la forma en la que se define la interfaz, es decir, los atributos y las operaciones que la conforman. Respecto a la información semántica, se suele optar por la definición del comportamiento individual de las operaciones de la interfaz.

Para la representación de este comportamiento, tradicionalmente se utilizan formalismos como ecuaciones algebraicas, pre/post condiciones o invariantes. Esta forma de representación detallada ofrece información sobre cuándo debe ser ejecutada una operación, cuál es el estado del componente tras su ejecución, etc. Además, lo ideal es que dicha información esté expresada en un lenguaje que pueda ser analizable de forma automática por un proceso software. Sin embargo, la gestión de los posibles emparejamientos entre interfaces, así como la gestión de combinaciones incorrectas (*mismatching*) [26] deriva en tiempos de cómputo aptos para el análisis de configuraciones en tiempo de diseño, pero inadmisibles para la construcción de configuraciones en tiempo de ejecución.

Nuestra propuesta de mediación semántica se basa en acotar los tipos posibles que se pueden utilizar para la descripción de la entrada y salida de las operaciones de las interfaces. Se crea, por tanto, un espacio de nombres que agrupa todos los tipos, que serán identificados de la forma `trader:nombreDelTipo`. Equivalen a tipos de datos complejos que proporcionan: a) información sobre los datos que componen el tipo de datos complejo y b) información acerca de qué operación utiliza este tipo de dato y de si se utiliza como entrada o salida de la operación.

Supongamos una operación nombrada como `loadLayer` perteneciente a la interfaz `manageLayers` del componente `mapa` de la Figura 2. Esta operación no define salida (`Output` en el metamodelo) y su entrada (`Input`) está descrita por el tipo `trader:loadLayerInput`. Dicho tipo está compuesto a su vez por dos elementos: `layerID` que es de tipo *String* y `serviceOGC` que es del tipo *anyURI*. Con esta información, podemos identificar qué componentes hacen uso de operaciones que puedan emparejarse con la anterior.

El conjunto de tipos definidos conforman un vocabulario de tipos que puede ser, además, organizado en una ontología para la gestión de sinónimos. Esta decisión presenta limitaciones (respecto a poder utilizar cualquier tipo de dato como entrada o salida de las operaciones) puesto que obliga a los desarrolladores de componentes a conocer y utilizar los tipos existentes. Sin embargo, es la solución óptima para poder realizar comparaciones de componentes teniendo en cuenta información semántica de sus operaciones y que este proceso se ejecute en tiempos asumibles para la generación de las arquitecturas en tiempo de ejecución. Queda por precisar que, el emparejamiento que se realiza a partir de esta información, no efectúa ningún cálculo relacionado con el orden de ejecución de las operaciones de un componente. Este orden queda inherente a la lógica de negocio que implementa cada componente, y se asume que los componentes que hacen uso de operaciones de otros componentes (dependencia entre interfaz requerida y proporcionada) llaman a las operaciones en el orden correcto.

Además de este tipo de descripción de las interfaces, la semántica está presente en la evaluación de componentes y arquitecturas que se utiliza en la función heurística del algoritmo A*, como se podrá ver en la subsección siguiente.

3.3 Algoritmo A* para la resolución de configuraciones

El cálculo y la construcción de las configuraciones de componentes concretos se lleva a cabo por el módulo *Configs* de nuestro proceso de mediación. En él se hace

uso de un algoritmo de búsqueda tipo A*. En este tipo de algoritmos, un grafo representa el espacio de búsqueda y sus nodos identifican los estados por los que se puede navegar dentro de dicho espacio. El objetivo es encontrar el camino de menor coste desde un nodo inicial al objetivo. Para el cálculo del coste, se utiliza una función de evaluación $f(x) = g(x) + h'(x)$. La función $g(x)$ representa una distancia conocida (pre-calculada) entre el nodo inicial y el nodo actual. Por otro lado, $h'(x)$ identifica el valor estimado de una heurística admisible ($h(x)$) de la distancia del nodo actual al nodo objetivo. Para que sea admisible, la heurística no debe sobrestimar el valor real de la distancia que se calcula.

Este tipo de algoritmo siempre encuentra una solución si existe. Además, el proceso de búsqueda no necesariamente debe explorar todos los nodos del grafo para encontrar dicha solución. Su complejidad depende de la calidad de la heurística definida. En el peor de los casos, el orden es exponencial; mientras que en el caso mejor (en el que la heurística estimada se acerca a la heurística óptima) el orden de complejidad es lineal. Este es el principal motivo de la elección de este tipo de algoritmo, puesto que en un algoritmo de búsqueda exhaustivo para la construcción de configuraciones [7], el orden siempre es exponencial y siempre se deben calcular la heurística para todos los nodos del especio de búsqueda.

El hecho de que el cálculo de las configuraciones de arquitecturas concretas se realice en tiempo de ejecución, es otro de los motivos para escoger este tipo de algoritmo. El camino de exploración siempre avanza hacia una solución cuya distancia respecto al nodo objetivo es siempre menor al estado anterior. Por tanto, podemos tener una referencia a la última "mejor solución" y hacer uso de ella si debemos forzar la finalización de la búsqueda (por ejemplo, por limitaciones de tiempo o alguna otra restricción de entrada).

En nuestra propuesta, los nodos del grafo representan configuraciones de componentes concretos de forma que, un nodo es adyacente a otro si su configuración difiere en un componente. Para limitar el grafo de búsqueda, previamente a la ejecución del algoritmo, se ha realizado una agrupación de los nodos candidatos para resolver una arquitectura (Figura 4). Cada grupo se corresponde con un componente de la arquitectura abstracta y contiene aquellos componentes concretos con alguna operación (de las interfaces proporcionadas) en común. De esta manera, en el grafo no existen nodos con componentes del mismo grupo.

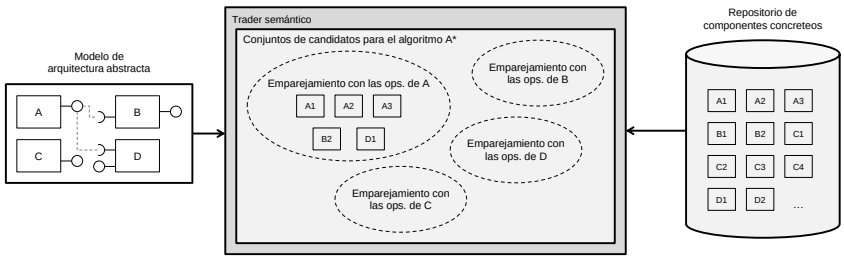

Fig. 4. Agrupación de candidatos en función de las operaciones de las interfaces

El Algoritmo 1 muestra el pseudocódigo del proceso desarrollado. Sin entrar en descripciones pertenecientes al funcionamiento de un algoritmo A* genérico, se van a aclarar algunos apectos de nuestra propuesta. El algoritmo comienza con un nodo inicial (*source*) que es adyacente a todos los nodos que se crean a partir de los componentes de uno de los grupos (Figura 4). Esos nodos son los únicos existentes en el grafo y el resto de nodos se crean de forma dinámica (línea 25 del Algoritmo 1) cuando un nuevo nodo es explorado. Además, sólo se crean nuevos vecinos si las configuraciones creadas no exceden el tamaño de la definición abstracta (línea 22). Estas optimizaciones limitan el espacio de búsqueda del algoritmo y reduce el total de nodos sobre los que hay que realizar cálculos de la función de evaluación $f(x)$.

Dicha función es la que se utiliza como referencia para la ordenación de la cola de prioridad que almacena el conjunto de nodos "abiertos" y a partir de la

Algoritmo 1 A* para la búsqueda de configuraciones de componentes

```
1: function aStar(source, AAM)
2:     openSet ← {source} // Conjunto de nodos para su evaluación
3:     pQueue ← {source} // Cola de prioridad para el conjunto de nodos anterior
4:     closeSet ← ∅ // Conjunto de nodos que ya han sido evaluados
5:     discardedConfigs ← 0 // Contador del nº de configuraciones descartadas
6:     notDesiredCC ← ∅ // Almacena los candidatos no deseados
7:     firstSolution ← false // Determina cuándo se encuentra la 1ª solución
8:     bestNode ← ∅ // Representa el mejor nodo del grafo
9:     while openSet ≠ ∅ do
10:        currentNode ← pQueue.poll()
11:        if currentNode.getH() < bestNode.getH() then bestNode ← currentNode
12:        end if
13:        if currentNode.getH() == 0 then
14:            if firstSolution == false then firstSolution ← true
15:            end if
16:            bestNode ← currentNode
17:            if evaluateCAM(currentNode, AAM) == true then return bestNode
18:            else discardedConfigs ← discardedConfigs + 1
19:            end if
20:        else
21:            closeSet.put(currentNode)
22:            if checkCAMSize(currentNode) == true then
23:                if contains(notDesiredCC, currentNode) == false then
24:                    neighbors = ∅ // Conjunto de vecinos al nodo actual
25:                    neighbors ← createNewAdajectNodes(currentNode)
26:                    for each neighbor in neighbors do
27:                        if contains(closeSet, neighbor) == false then
28:                            if contains(openSet, neighbor) == false then
29:                                h ← heuristics(neighbor, AAM)
30:                                newNode ← createNode(neighbor, h)
31:                                if h == 0 then ... // (líneas 14–19 del algoritmo)
32:                                else
33:                                    openSet.put(newNode)
34:                                    pQueue.add(newNode)
35:                                end if
36:                            end if
37:                        end if
38:                    end for
39:                end if
40:            end if
41:        end if
42:    end while
43:    return bestNode
44: end function
```

cual se selecciona cada nuevo nodo a explorar (línea 10). Por defecto, el valor de $g(x)$ es 1, puesto que un nodo difiere del otro en la incorporación de un nuevo componente concreto. No obstante, tras la depuración del algoritmo, cuando $g(x) = 0$ se alcanzan soluciones en menor tiempo, haciendo que el algoritmo A* equivalga a un algoritmo voraz (*greedy*). Por ello, se ha establecido el valor $g(x)$ como configurable a través de la interfaz *Admin*.

Por otro lado, el valor de $h'(x)$ (línea 29) representa la distancia de la configuración de componentes concretos asociada al nodo actual con respecto a la arquitectura abstracta de entrada (AAM). Por motivos de optimización, dicha distancia está calculada únicamente a partir de la información semántica de la parte funcional de los componentes. Esta decisión asegura la resolución de una configuración válida respecto a la parte funcional en tiempos menores que si se evalúan todas las partes de todos los componentes. No obstante, cuando se encuentra una configuración que cumple con la parte funcional (líneas 13 y 31), se realiza una evaluación completa de la configuración calculando la distancia con respecto al AAM utilizando todas las partes del componente. En esta evaluación se comprueba también: a) que las configuraciones son cerradas (no presentan componentes con interfaces requeridas adicionales a la arquitectura abstracta) y b) que cumplen con la definición de entrada (la funcionalidad está agrupada en los componentes como determina la arquitectura abstracta).

En la configuración de nuestro proceso de mediación, también se pueden configurar las distancias mínimas establecidas para considerar que una configuración resuelve una arquitectura. Por ejemplo, a través de la interfaz *Admin*, se puede establecer que se debe cumplir con un ratio de 0.8 en la parte de las propiedades de los componentes, determinando así que la distancia para esa parte específica se encuentre en el intervalo $[0, 0.2]$.

4 Caso estudio: Una aplicación a las interfaces *mashups*

En esta sección se presenta la aplicación de la propuesta al domino de las interfaces *mashups* [11]. En particular, el escenario de ejemplo es el mencionado anteriormente de una GUI para la gestión de un SIG (Figura 2). En ella se pueden visualizar componentes de los siguientes tipos: mapa, histograma, leyenda, cabecera, almacenamiento, audio, vídeo, grabación, redes sociales y lector de RSS. Las arquitecturas que conforman *mashups* de este dominio, presentan dependencias entre sus componentes, por ejemplo, el componente de grabación requiere un componente de vídeo y otro de audio; el componente de redes sociales requiere información de usuario proporcionada por el de cabecera; mientras que el lector de RSS es un componente aislado. No se explica la finalidad ni especificación de cada uno de estos componentes, puesto que el fin es crear un escenario de tamaño suficiente para la validación de la propuesta.

Para cada uno de estos tipos de componentes, *i.e.*, para cada componente abstracto, se han definido 20 especificaciones de componentes concretos. Esto supone un total de 200 componentes candidatos ($10 * 20$) en el caso de arquitecturas donde estén presentes los diez tipos de componentes. La agrupación de

candidatos descrita en la Figura 4, limita el espacio de búsqueda a un número combinario de $\binom{20}{10} = 184756$ configuraciones. El conjunto de componentes candidatos se encuentra insertado dentro de un repositorio con otros 10000 especificaciones de componentes concretos que han sido generados de forma aleatoria.

Dentro de este escenario, se han ejecutado pruebas para la resolución de arquitecturas de diferentes tamaños, desde 1 hasta 10. Para cada tamaño de arquitectura, se ha ejecutado 100 veces el proceso y se ha calculado la media obtenida para los siguientes datos: tiempo total para la obtención de la solución óptima, tiempo en el que se obtiene la primera solución funcional y número de configuraciones descartadas. Estas pruebas fueron ejecutadas bajo un *framework* Eclipse luna-SR2 en una máquina con 3.33 GHz Intel(R) Core(TM) i5 de procesador y 8 GB de memoria principal.

Los datos de la primera gráfica de la Figura 5 pueden aproximarse con una progresión lineal del tiempo a medida que aumentamos el número de componentes de la arquitectura. Los datos de la segunda gráfica ofrecen una garantía de lo fiable que es el proceso para la obtención de soluciones que al menos han sido emparejadas en la parte funcional (su tendencia también es lineal a medida que aumenta el tamaño de la arquitectura). En la primera solución funcional y la solución final, se descartan configuraciones no válidas, llegado hasta valores de 350 configuraciones descartadas para arquitecturas con 10 componentes.

Es necesario puntualizar que la configuración del proceso de mediación para estas pruebas es la más restrictiva posible, es decir, la distancia de emparejamiento calculada en el algoritmo debe ser 0 para todas las partes de los componentes de la arquitectura. Este hecho supone maximizar el tiempo total en la obtención de una arquitectura final, con el objetivo de validar el proceso de forma correcta. Los tiempos más altos obtenidos de 1.2 segundos para interfaces de usuario con 10 componentes son tiempos aceptables. Más aún cuando en este tipo de interfaces de usuario *mashups*, no suelen presentarse a la vez en la arquitectura un gran número de componentes (al ser componentes de una granulidad media/alta que encapsulan funcionalidad de mini-aplicación). Sin embargo, estos tiempos deben ser mejorados en el caso de otros entornos que requieran una mayor velocidad en el proceso de re-configuración de las arquitecturas, por ejemplo, en arquitecturas software de robótica cuando se ejecutan tareas de riesgo.

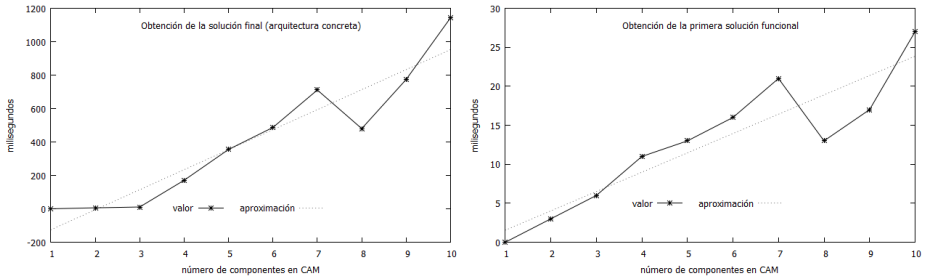

Fig. 5. Resultados de la validación del caso estudio

5 Trabajo relacionado

Uno de los elementos principales de nuestra propuesta es la utilización de componentes tipo COTS para la construcción de arquitecturas. En dicha construcción, intervienen procesos de selección y evaluación de componentes [28]. Un ejemplo de trabajo en el que se abordan estos procesos es *Off-The-Shelf Option* (OTSO) [24], donde un criterio de evaluación jerárquica analiza las características de los componentes basándo además en otros factores como la infraestructura de la organización o la disponibilidad de librerías. En [17], el sistema DesCOTS propone una metodología basada en un modelo de calidad que divide las características de los componente para su evaluación.

El trabajo presentado en [32] evalúa los componentes y realiza un ranking en términos de su rendimiento y atendiendo a múltiples criterios. Otras propuestas como [14] realizan una gestión de las dependencias entre componentes haciendo uso de modelo orientados objetivos, como base para la selección de componentes. En [6] se describe una propuesta de selección de componentes COTS en repositorios de gran tamaño haciendo uso del concepto de "integrador" en lugar de un proceso de mediación. En cualquier caso, a diferencia de nuestra propuesta, ninguno de los trabajos mencionados dan soporte a la selección de componentes o el cálculo de configuraciones en tiempo de ejecución.

Por otro lado, el trabajo descrito en [22] es la base de nuestro proceso de regeneración. En él se describe un servicio de mediación para la gestión de componentes COTS y la construcción de configuraciones en tiempo de diseño. Nuestra propuesta se inspira en su modelo de documentación de componentes para conseguir la generación de arquitecturas en tiempo de ejecución. Respecto a los modelos de documentación, además de [22], existen múltiples propuestas para la caracterización de componentes COTS, siendo algunos ejemplos [29] y [31]. Nuestra propuesta realiza la caracterización y validación de los componentes y las arqutiecturas haciendo uso de técnicas MDE, e incorpora información semántica de forma simplificada para su análisis en tiempo de ejecución.

Otras propuestas como [33] presentan un método semi-automático para la identificación y clasificación de componentes basándose en un taxonomía y en información semántica de entrada. En [3] también se señalan las taxonomías y las ontologías como opción para aportar información semántica en la identificación de COTS. En nuestro caso, el establecimiento de un vocabulario de tipos que pueden utilizarse para describir las propiedades de los componentes, hace posible la construcción de una clasificación de este tipo. Para la exploración de las posibles soluciones, una opción era utilizar algoritmos heurísticos que evaluaran las configuraciones [25]. Finalmente, se decidió escoger un algoritmo A* genérico y realizar algunas modificaciones para la optimización del proceso y que fuera posible la construcción de configuraciones en tiempo de ejecución.

6 Conclusiones y trabajo futuro

Este artículo presenta una propuesta para la generación de arquitecturas en tiempo de ejecución. En particular, se centra en la descripción del proceso de

regeneración que forma parte de una metodología para la adaptación de arquitecturas software. Dicho proceso se encargada de construir arquitecturas concretas a partir de sus definiciones de arquitectura abstracta. Para ello, se ha desarrollado un proceso de mediación que gestiona los componentes y las arquitecturas, y que se encarga de construir las configuraciones en tiempo de ejecución.

El cálculo de las configuraciones se realiza a través de un algoritmo de búsqueda tipo A* que hace uso de la información semántica y sintáctica de los componentes para 1) seleccionar el conjunto de componentes de candidatos y 2) para evaluar las distintas configuraciones a través de una heurística definida. Para la validación de la propuesta, se ha desarrollado un caso de estudio aplicado al dominio de interfaces gráficas de usuario tipo *mashups*.

Como trabajo futuro, se investigará acerca de las posibles mejoras en el rendimiento que pueden ser aplicadas en el algoritmo A* propuesto, por ejemplo, paralelizando parte de la ejecución o incorporando nuevas técnicas para el cálculo de la distancia entre componentes y arquitecturas. También es necesario desarrollar un mecanismo para facilitar la gestión de los tipos de componentes, así como los tipos de entrada y salida de las operaciones de las interfaces. Por otro lado, los tiempos de ejecución deben ser mejorados en el caso de la aplicación de la propuesta en otros dominios.

Agradecimientos. Este trabajo ha sido desarrollado bajo el marco del proyecto TIN2013-41576-R del Ministerio de Economía y Competitividad (MINECO) y de fondos EU ERDF, bajo un beca FPU (AP2010-3259), y bajo el marco del proyecto P10-TIC-6114 de La Junta de Andalucía.

Referencias

1. Cabot, J., Gogolla, M.: Object Constraint Language (OCL): A definitive guide. LNCS 7320, pp. 58–90 (2012)
2. Carney, D., Leng, F.: What Do You Mean by COTS? Finally, a Useful Answer. IEEE Software, 17(2), pp. 83–86 (2000)
3. Cechich, A., *et al.*: Trends on COTS component identification. 5th COTS-BS Systems, pp. 90–99. IEEE (2006)
4. Cetina, C., *et al.*: Autonomic computing through reuse of variability models at runtime: The case of smart homes. Computer, 42(10), pp. 37–43. IEEE (2009)
5. Chung, L., Cooper, K.: Matching, ranking, and selecting components: A cots-aware requirements engineering and software architecting approach. In: MPEC Workshop of 26th Int. Conf. on Software Engineering, pp. 41–44 (2004)
6. Clark, J., *et al.*: Selecting components in large COTS repositories. Journal of Systems and Software, 73(2), 323–331 (2004)
7. Criado, J., Iribarne, L., Padilla, N.: Resolving Platform Specific Models at runtime using an MDE-based Trading approach. LNCS 8186, pp. 274–283 (2013)
8. Criado, J., *et al.*: Toward the adaptation of component-based architectures by model transformation: behind smart user interfaces. Software: Practice and Experience. doi:10.1002/spe.2306 (2014)
9. Crnkovic, I.: Component-based Software Engineering – New Challenges in Software Development. Software Focus, 2(4), 127–133 (2001)

10. Crnkovic, I.: Component-based Software Engineering for Embedded Systems. 27th ICSE, pp. 712–713. ACM (2005)
11. Daniel, F., Matera, M.: Mashups - Concepts, Models and Architectures. Data-Centric Systems and Applications. Springer. ISBN 978-3-642-55048-5 (2014)
12. Dijkman, R., Dumas, M., García-Bañuelos, L.: Graph matching algorithms for business process model similarity search. LNCS 5701, pp. 48–63 (2009)
13. Fouquet, F., *et al.*: A dynamic component model for cyber physical systems. In: 15th ACM SIGSOFT, pp. 135–144. ACM (2012)
14. Franch, X., Maiden, N.A.: Modelling component dependencies to inform their selection. COTS-Based Software Systems, pp. 81–91. Springer (2003)
15. Garlan, D., *et al.*: Rainbow: Architecture-based self-adaptation with reusable infrastructure. Computer, 37(10), pp. 46–54. IEEE (2004)
16. Graham, S., *et al.*: Building Web services with Java: making sense of XML, SOAP, WSDL, and UDDI. SAMS publishing (2004)
17. Grau, G., *et al.*: DesCOTS: a software system for selecting COTS components. 30th Euromicro Conf., pp. 118–126. IEEE (2004)
18. Grundy, J., Hosking, J.: Developing adaptable user interfaces for component-based systems. Interacting with Computers, 14(3), pp. 175–194 (2002)
19. Gui, N., De Florio, V., Holvoet, T.: Transformer: an adaptation framework supporting contextual adaptation behavior composition. SPE, 43(8), pp. 937–967 (2013)
20. Hoffner, Y., Schade, A.: Co-operation, contracts, contractual match-making and binding. EDOC'98, pp. 75–86. IEEE. (1998)
21. Iribarne, L., Troya, J.M., Vallecillo, A.: Selecting software components with multiple interfaces. 28th Euromicro Conf., pp. 26–32. IEEE (2002)
22. Iribarne, L., Troya, J.M., Vallecillo, A.: A trading service for COTS components. The Computer Journal, 47(3), 342–357 (2004)
23. ISO/IEC 13235-1, ITU-T X.950. Information Technology – Open Distributed Processing – Trading function: Specification (1998)
24. Kontio, J., Caldiera, G., Basili, V.R.: Defining factors, goals and criteria for reusable component evaluation. In: CASCON 1996, pp. 21–32. IBM Press (1996)
25. Korf, R.E.: Real-time heuristic search. Artificial Int., 42(2), pp. 189–211 (1990)
26. Li, X., Fan, Y., Jiang, F.: A classification of service composition mismatches to support service mediation. 6th Int. Conf. on GCC, pp. 315–321. IEEE (2007)
27. Merz, M., Muller, K., Lamersdorf, W.: Service trading and mediation in distributed computing systems. 14th Int. Conf. on DCS, pp. 450–457. IEEE (1994)
28. Mohamed, A., Ruhe, G., Eberlein, A.: COTS selection: past, present, and future. Engineering of Computer-Based Systems, pp. 103–114. IEEE (2007)
29. Morisio, M., Torchiano, M.: Definition and classification of COTS: a proposal. COTS-Based Software Systems, LNCS 2255, pp. 165–175 (2002)
30. Rodríguez-Gracia, *et al.*: Runtime adaptation of architectural models: an approach for adapting user interfaces. LNCS 7602, pp. 16–30 (2012)
31. Sassi, S.B., Jilani, L.L., Ghezala, H.H.B.: COTS characterization model in a COTS-based development environment. 10th AP-SE Conf., pp. 352–361. IEEE (2003)
32. Shyur, H.J.: COTS evaluation using modified TOPSIS and ANP. Applied Mathematics and Computation, 177(1), 251-259 (2006)
33. Sjachyn, M., Beus-Dukic, L.: Semantic component selection-SemaCS. In: 5th ICCBSS, pp. 1–7. IEEE (2006)
34. Vallecillos, J., *et al.*: Dynamic Mashup Interfaces for Information Systems Using Widgets-as-a-Service. LNCS 8842, pp. 438–447 (2014)