Evolución arquitectónica de servicios basada en modelos CVL con cardinalidad

José Miguel Horcas, Mónica Pinto, and Lidia Fuentes

Universidad de Málaga, Andalucía Tech, Spain
{horcas,pinto,lff}@lcc.uma.es, http://caosd.lcc.uma.es/

Resumen La computación en la nube se está convirtiendo en un mecanismo predominante para desplegar fácilmente aplicaciones con requisitos especiales, tales como el almacenamiento masivo compartido, o el equilibrado de carga. Esta funcionalidad se proporciona normalmente como servicios por las plataformas en la nube. Un desarrollador puede mejorar tanto el despliegue de sus aplicaciones como la productividad siguiendo un enfoque *multi-tenancy*, donde diferentes variantes de la misma aplicación pueden adaptarse rápidamente a las necesidades de cada usuario (*tenant*). Sin embargo, gestionar la variabilidad inherente a las aplicaciones multi-tenant, con cientos de usuarios y miles de configuraciones arquitectónicas diferentes, puede llegar a ser una tarea intratable de abordar manualmente. En este artículo, se propone un enfoque de línea de producto software en el cual: (1) usamos modelos de variabilidad con cardinalidad para modelar cada tenant como una característica clonable, (2) automatizamos el proceso de evolución de las arquitecturas de aplicaciones multi-tenant, y (3) demostramos que la implementación de los procesos de evolución es correcta y eficiente para un número elevado de tenants en un tiempo razonable.

Keywords: Cardinalidad, Evolución, Línea de Producto Arquitectónica, Variabilidad, CVL

1. Introducción

La computación en la nube se está convirtiendo en el principal mecanismo para desplegar fácilmente aplicaciones con requisitos especiales, tales como el almacenamiento masivo compartido, escalado automático, o el equilibrado de carga [2]. Los desarrolladores pueden integrar los servicios ofrecidos por las plataformas en la nube (e.g., Microsoft Azure, Amazon Web Services) como parte de la arquitectura de sus aplicaciones, disminuyendo así el tiempo de desarrollo.

Diferentes versiones de la misma aplicación pueden ser desarrolladas siguiendo un enfoque *multitenancy* [12], donde cada variante de una aplicación puede ser personalizada según las necesidades de cada usuario (*tenant*). Sin embargo, gestionar la variabilidad inherente en las aplicaciones multi-tenant, donde es necesario mantener diferentes configuraciones de la arquitectura software para cada tenant, no es una tarea sencilla. Las Líneas de Producto Software (SPL, del inglés *Software Product Line*) [15] constituyen un enfoque ampliamente usado para especificar la variabilidad en general, y específicamente en arquitecturas orientadas a servicios [5]. Es aquí dónde las aplicaciones multi-tenant plantean

un nuevo reto: representar de forma explícita y gestionar la existencia de configuraciones simultáneas de los mismos servicios, uno para cada tenant [7].

Además, se debe tener en cuenta la gestión de la evolución de este tipo de aplicaciones. Los proveedores de las plataformas en la nube están continuamente evolucionando y actualizando sus tecnologías con el fin de ser competitivos en el mercado. Por otra parte, la funcionalidad específica de la aplicación puede evolucionar para tener en cuenta nuevas características solicitadas por los usuarios. En ambos casos, la arquitectura software de la aplicación en la nube debe ser adaptada para añadir nuevos componentes software o eliminar y reconfigurar los existentes. Sin embargo, gestionar la evolución de una aplicación multi-tenant con cientos de usuarios y miles de configuraciones posibles puede llegar a ser una tarea inabordable manualmente. Aunque existen SPLs en el contexto de las aplicaciones multi-tenant [5,9,14,19], la mayoría presenta dos principales limitaciones: (i) no tienen en cuenta la automatización de la evolución a nivel de la arquitectura software; y (ii) instancian la SPL de forma individual para cada tenant. Esto dificulta la realización de los cambios de forma automática y consistente, y aumenta la complejidad de cambiar simultáneamente las configuraciones arquitectónicas existentes para cada tenant.

Los principales objetivos de este artículo son: (1) identificar los cambios necesarios en las aplicaciones cuando los requisitos, tanto de la propia aplicación como de los servicios proporcionados por la plataforma en la nube, cambian; y (2) obtener de manera simultánea, para cada tenant, una arquitectura software evolucionada que sea válida y consistente con la configuración actualmente desplegada en ese tenant. Para lograr estos objetivos se propone una línea de productos arquitectónica (PLA) [4] en la que: (i) modelamos la configuración de cada tenant como una característica clonable usando modelos de variabilidad con cardinalidad en CVL [10]; (ii) automatizamos el proceso de evolución de la arquitectura multi-tenant definiendo tres algoritmos que propagan automáticamente los cambios necesarios en la configuración arquitectónica desplegada en cada tenant; y (iii) demostramos que los algoritmos son correctos y eficientes para un número elevado de tenants. Ilustramos nuestra propuesta con una aplicación en el dominio del software médico.

El artículo se organiza de la siguiente manera. La Sección 2 describe los retos principales de nuestra propuesta a través de un caso de estudio. La Sección 3 explica como modelar la variabilidad de las aplicaciones multi-tenant con CVL. La Sección 4 y 5 detallan nuestro proceso de evolución y los algoritmos propuestos. La Sección 6 evalúa nuestra propuesta. Finalmente, la Sección 7 discute el trabajo relacionado y la Sección 8 las conclusiones y el trabajo futuro.

2. Motivación y caso de estudio

Nuestro caso de estudio es una aplicación para la gestión y administración de servicios médicos en hospitales. Con el fin de ahorrar en costes de implementación y mantenimiento, se decide desarrollar la aplicación usando una plataforma en la nube (e.g., Microsoft Azure) [18]. El objetivo es vender la aplicación médica a diferentes clientes (hospitales), por lo que para tener configuraciones diferentes para cada hospital, la aplicación seguirá un enfoque multi-tenant, considerando

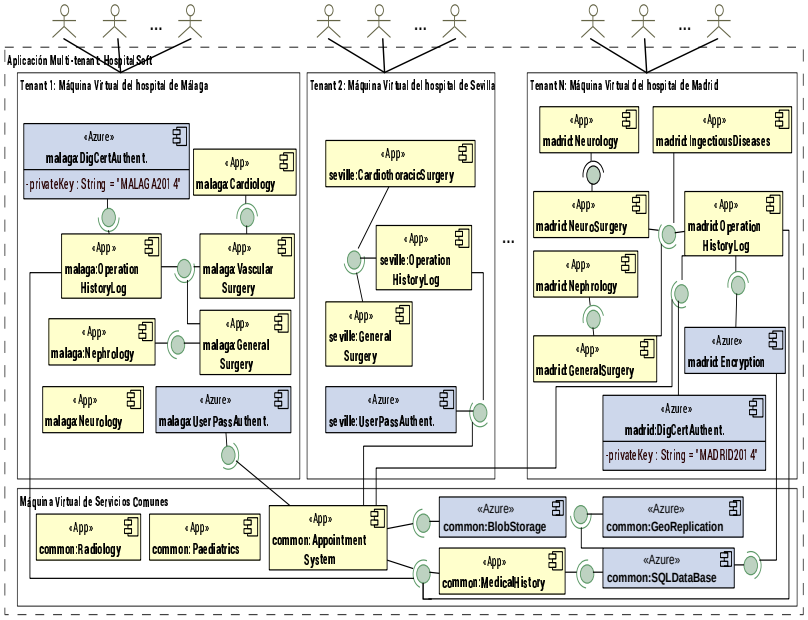

Figura 1. Arquitectura software de una aplicación multi-tenant en Microsoft Azure.

cada hospital como un tenant. Además, se plantea utilizar varios de los servicios proporcionados por la plataforma Azure, como la persistencia y la geo-replicación de datos. Por lo tanto, la aplicación multi-tenant tendrá un conjunto de servicios comunes disponibles desde la misma máquina virtual y compartidos por todos los usuarios, y también un conjunto de servicios específicos para cada usuario disponibles a través de máquinas virtuales específicas para cada usuario. La Figura 1 muestra una instancia de la arquitectura de la aplicación médica instanciada y configurada para tres usuarios: los hospitales de Málaga, Sevilla y Madrid.

Primero, se proporciona un conjunto de servicios que son comunes para todos los tenants desde la misma máquina virtual (`Máquina Virtual de Servicios Compunes`). Algunos de ellos, estereotipados como `«App»`, son específicos de la aplicación médica, como el sistema de citas (componente `Appointment System`) y el historial médico de los pacientes (`Medical History`). Otros, estereotipados como `«Azure»`, son servicios ofrecidos por la plataforma de Microsoft, como el servicio de persistencia para almacenar los datos clínicos (`SQL DataBase`) y las citas de los pacientes (`BlobStorage`), o la posibilidad de crear múltiples copias de la información en diferentes centros de datos (`GeoReplication`). Segundo, además de estos servicios comunes, la aplicación proporciona para cada tenant un conjunto de servicios específicos configurados acorde a sus diferentes necesidades. Por ejemplo, el hospital de Málaga es el único que realiza cirugías vasculares, por lo que el componente `VascularSurgery` está incluido únicamente en este tenant. Análogamente, los componentes `Nephrology` y `Neurology` están instanciados para el hospital de Málaga y de Madrid, pero no para el hospital de Sevilla. Finalmente, no sólo los componentes específicos de la aplicación varían entre los diferentes tenants, algunos servicios de la plataforma en la nube también pue-

den ser configurados por cada tenant. Por ejemplo, el método de autenticación es diferente para cada hospital: el hospital de Málaga usa un certificado digital para autenticar al personal médico en el sistema (componente `DigCertAuthent`) y nombre de usuario y contraseña para autenticar a los pacientes en el servicio de cita médica en línea (`UserPassAuthent`); mientras que el hospital de Sevilla usa autenticación mediante el nombre de usuario y contraseña para todos indistintamente, y el hospital de Madrid usa certificado digital para todos.

En las aplicaciones multi-tenant hay características (implementadas como componentes) que son requeridas por todos los tenants y otras que son variables y configurables. Esto significa que normalmente sólo un subconjunto de las características variables de la aplicación son desplegadas en cada tenant. Teniendo en cuenta que se deben generar y mantener cientos de configuraciones diferentes de la aplicación, lo que implica la gestión de miles de componentes, un reto importante es *proporcionar mecanismos para representar y gestionar directamente la variablidad de las aplicaciones multi-tenant.*

Pero una vez desplegadas, las aplicaciones multi-tenant tienen que ser adaptadas a mejoras tecnológicas y a cambios en las necesidades de los usuarios. Por ejemplo, nuevos métodos de autenticación (e.g., autenticación biométrica, autenticación usando redes sociales) o persistencia (e.g., fragmentación de base de datos, grupos de afinidad) aparecen con frecuencia. Si se quiere proporcionar estos nuevos servicios a los clientes, éstos se deben incorporar a los tenants que lo requieran. También se podría cambiar el proveedor de la plataforma en la nube y migrar la aplicación a una nueva (ej: Amazon Web Services). En este caso, la parte de la arquitectura de la aplicación que depende de los servicios de la plataforma tiene que adaptarse a los servicios que ofrece la nueva plataforma en la nube. Por último, durante la vida de la aplicación, los clientes pueden exigir nuevas funcionalidades (e.g., un nuevo módulo para la gestión de transplantes o cambios en los métodos de autenticación para identificar a los pacientes).

Considerando los cambios que es necesario realizar en la arquitectura software de la aplicación multi-tenant, todas las situaciones mencionadas anteriormente se pueden representar con tres escenarios diferentes de evolución: (1) un nuevo componente necesita ser incorporado en la arquitectura software, ya sea proporcionado por la plataforma en la nube (Azure) o implementado por el desarrollador de la aplicación; (2) un componente existente necesita ser eliminado de la arquitectura software; y (3) un componente existente necesita ser reconfigurado con nuevos parámetros. Sin embargo, la evolución de una aplicación multi-tenant implica también tener que calcular y realizar estos cambios para miles de componentes que se ejecutan en cientos de tenants, convirtiendo el proceso de evolución en una tarea intratable de abordar manualmente. Por lo tanto, otros retos importantes de la evolución en aplicaciones multi-tenant son *el cálculo automático de los cambios que se deben realizar en cada tenant* y *la propagación automática de estos cambios para todos los tenant a nivel arquitectónico.* Por otra parte, este proceso de evolución automático sólo será útil si es correcto y eficiente para un gran número de tenants, por lo que necesitamos *demostrar la eficiencia y la corrección del proceso de evolución.*

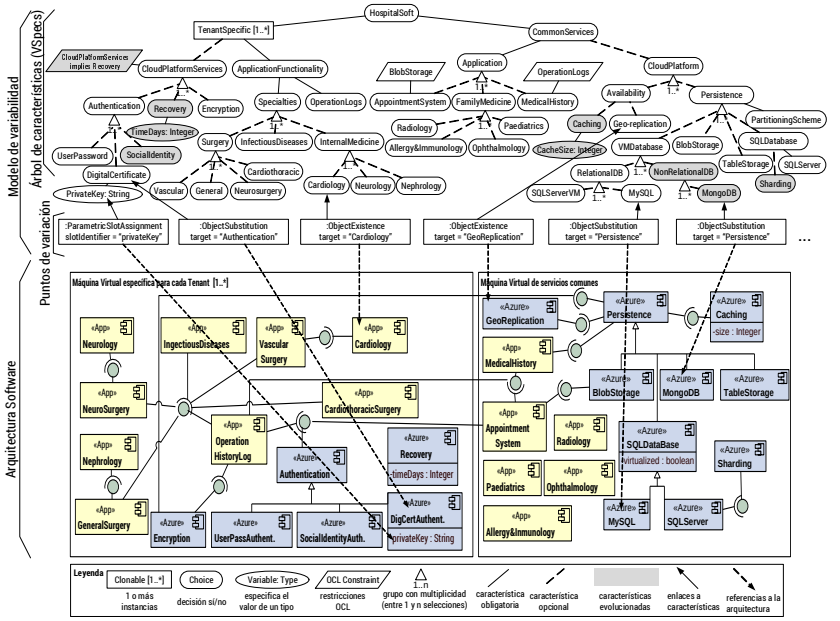

Figura 2. Modelo de variabilidad en CVL y arquitectura software.

3. Gestión de la variabilidad con CVL

En esta sección explicamos cómo nuestra propuesta usa CVL (*Common Variability Language*) [10] para gestionar la variabilidad de la arquitectura software de todos los tenant en una aplicación basada en la nube. CVL es un lenguaje independiente del dominio para especificar y resolver la variabilidad en modelos basados en MOF.[1] Concretamente, como muestra la Figura 2 para nuestro caso de estudio, modelamos explícitamente la variabilidad de las características que son específicas de cada tenant (i.e., sub-árbol `TenantSpecific[1..*]`) y la funcionalidad común que comparten todos los tenants (i.e., sub-árbol `CommonServices`).

El **modelo de variabilidad** CVL está formado por dos partes. La primera es una parte abstracta que modela las características opcionales y obligatorias (*VSpecs* en CVL, de *Variability Specifications*) y las restricciones entre ellas (*cross-tree constraints*). Esta parte abstracta se define mediante un árbol como el mostrado en la parte superior de la Figura 2 y especifica la funcionalidad de la aplicación y los servicios ofrecidos por la plataforma en la nube. La segunda parte del modelo de variabilidad son los puntos de variación (*variation points*), que aparecen en la parte central de la Figura 2. Cada punto de variación está asociado a una característica del árbol y tiene una o más referencias a elementos de la arquitectura software. Estos puntos de variación representan modificaciones específicas a realizar en la arquitectura (i.e., transformaciones modelo a modelo, M2M) cuando una característica ha sido seleccionada en una configuración concreta del modelo de variabilidad. Cuando hablamos de definir una **configuración del modelo de variabilidad** nos referimos a seleccionar un conjunto

[1] `http://www.omg.org/mof/`

de características en el árbol que cumplan el conjunto de restricciones. Luego, el motor de ejecución de CVL es el encargado de ejecutar las transformaciones M2M asociadas a cada punto de variación según las características seleccionadas. En CVL una configuración del modelo de variabilidad recibe el nombre de **modelo de resolución** (*resolution model).*

Para dar soporte a diferentes configuraciones para cada tenant, nuestra propuesta define la funcionalidad específica de los tenant bajo una característica *clonable* (`TenantSpecific[1..*]` en la Figura 2). La característica clonable tiene una cardinalidad [1..*] que indica que esta característica puede ser instanciada una o más veces y todas sus sub-características pueden ser configuradas de forma diferente para cada instancia. En nuestro ejemplo, la cardinalidad representa el número de tenants, y cada instancia de `TenantSpecific[1..*]` define la configuración para ese tenant específico (por ejemplo para el hospital de Sevilla).

Seleccionando las características apropiadas bajo la característica clonable `TenantSpecific[1..*]`, y ejecutando CVL, nuestra propuesta genera una configuración de la arquitectura como la mostrada en la Figura 1, donde cada tenant está configurado de acuerdo a sus necesidades.

4. Gestión de la evolución con CVL

Una vez que se ha generado y desplegado una configuración arquitectónica adaptada a los requisitos de cada tenant, la aplicación multi-tenant es susceptible de evolucionar debido a mejoras tecnológicas y/o a cambios en las necesidades de los clientes. Volviendo a nuestra aplicación médica, supongamos que Microsoft incorpora nuevas funcionalidades a su plataforma: un servicio de recuperación de datos, un método de autenticación basado en Facebook y un mecanismo de persistencia. Supongamos también que el proveedor de la aplicación médica quiere proporcionar estos nuevos servicios a sus tenants (i.e., a sus hospitales).

El primer paso en el proceso de evolución es adaptar el modelo de variabilidad con nuevas características (aparecen en color gris en la Figura 2), y la arquitectura de la aplicación con nuevos elementos arquitectónicos (la parte inferior de la Figura 2 muestra la arquitectura ya evolucionada). Por ejemplo, las características `Recovery`, `SocialIdentity`, `Caching`, `MongoDB` y `Sharding` han sido incorporadas en el modelo de variabilidad con el fin de añadir el nuevo servicio de recuperación, el nuevo método de autenticación, y el nuevo mecanismo de persistencia (una nueva base de datos no relacional y un nuevo mecanismo de partición de base de datos). Así mismo, la arquitectura ha sido adaptada con los nuevos componentes `Recovery`, `SocialIdentityAuth` y `MongoDB`, entre otros.

El segundo paso en el proceso de evolución es calcular de manera automática y consistente los cambios que son necesarios realizar en todos los tenants que están actualmente desplegados (Figura 1). El objetivo es que esas configuraciones ya desplegadas satisfagan el nuevo modelo de variabilidad y los nuevos requisitos de la aplicación. Para automatizar esta tarea nuestra propuesta divide este segundo paso en dos partes: (i) modificar la configuración actual de todos los tenants, generando una nueva configuración evolucionada a partir del modelo de variabilidad previamente evolucionado (algoritmo *Evolve Configuration*); y (ii)

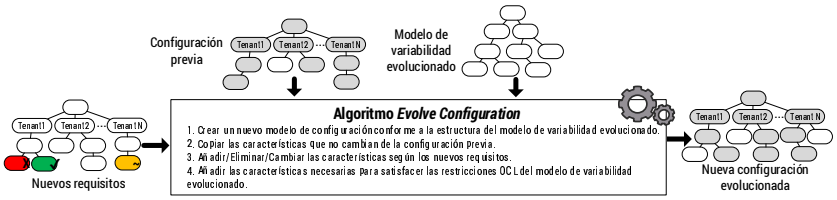

Figura 3. Algoritmo para evolucionar una configuración.

calcular las diferencias entre la configuración evolucionada y la configuración actualmente desplegada (algoritmo *Difference Configuration*). Recordamos que por configuración nos referimos a una instancia concreta del árbol de características.

El tercer paso es el más complicado en el proceso de evolución y consiste en propagar automáticamente los cambios previamente calculados a la arquitectura software actualmente desplegada. Para hacer esto, definimos un tercer algoritmo (*Create Weaving Model*) que genera un nuevo modelo en CVL que especifica cómo propagar a la arquitectura software las modificaciones definidas previamente a nivel de características (nos referimos a este modelo como *modelo de composición*). Este algoritmo es una de las principales contribuciones del artículo, debido a que las propuestas existentes que gestionan la evolución con SPLs (como en [1,8]) sólo abordan el problema de la evolución a nivel abstracto de características, y requieren modificar manualmente la arquitectura software para reflejar los cambios calculados — e.g., modificar el fichero de configuración de cada tenant, o definir manualmente un mapeo entre el modelo de configuración a nivel de características y la arquitectura evolucionada para cada tenant. Finalmente, CVL es ejecutado con el modelo de composición como entrada con el fin de obtener la arquitectura evolucionada con los cambios para cada tenant.[2]

4.1. Evolución del modelo de configuración

El algoritmo para evolucionar la configuración de una aplicación multi-tenant (*Evolve Configuration* en la Figura 3) recibe como entrada el modelo de la configuración actualmente desplegada (que será evolucionado), el modelo de variabilidad evolucionado (actualizado previamente por el proveedor de la aplicación), y la lista de las nuevas características requeridas por los clientes; y genera el modelo de la configuración evolucionada. El modelo generado representa, a nivel del árbol de características, una nueva configuración válida de la aplicación multi-tenant con todas las configuraciones de los tenants evolucionados.

La Figura 4 muestra una vista parcial del árbol de características donde, por limitación de espacio, se ha representado en el mismo árbol las tres entradas del algoritmo. El algoritmo genera un nuevo modelo donde, en primer lugar, se copian aquellas características que no cambian de la configuración previa (características en color blanco). A continuación, se añaden las nuevas características seleccionadas (marcadas con ✓), se omiten aquellas características que ya no son requeridas (marcadas con ×) y se añaden las características cuyos valores han cambiado (marcadas con ∼). Finalmente, se añaden aquellas características

[2] La formalización de los algoritmos y su definición completa está disponible en `http://caosd.lcc.uma.es/papers/evolutionAlgorithms.pdf`.

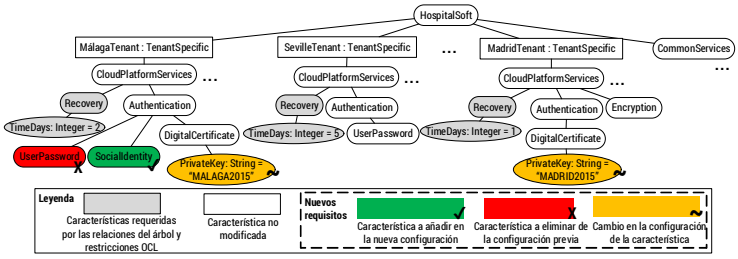

Figura 4. Información gestionada por el algoritmo *Evolve Configuration*.

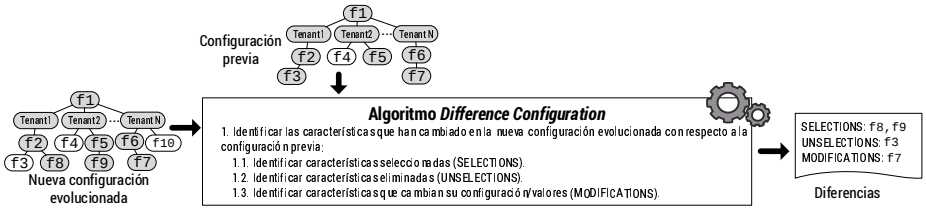

Figura 5. Algoritmo para calcular la diferencia entre dos configuraciones.

requeridas por las nuevas restricciones definidas en el modelo de variabilidad evolucionado (en gris). En nuestro ejemplo, para el tenant Málaga, la característica `UserPassword` es eliminada, mientras que `SocialIdentity` es añadida como nuevo requisito. Además, el parámetro `PrivateKey` del certificado digital es actualizado a un nuevo valor tanto para el tenant Málaga (‘‘`MALAGA2015`’’) como para el tenant Madrid (‘‘`MADRID2015`’’). También la característica `Recovery` y su parámetro `TimeDays`, que especifica el intervalo de copia de seguridad, son añadidos en todos los tenants debido a la nueva restricción (`CloudPlatformServices implies Recovery`) en el modelo de variabilidad evolucionado.

Una vez que el modelo de resolución evolucionado ha sido generado, el siguiente algoritmo calcula la diferencia entre la nueva y la anterior configuración.

4.2. Cálculo de las diferencias entre configuraciones

El algoritmo *Difference Configuration* recibe como entrada dos configuraciones (i.e., la configuración previa y la configuración nueva obtenida con el algoritmo *Evolve Configuration*) y calcula la diferencia entre ellas (Figura 5). Las diferencias están determinadas por: (1) las nuevas características seleccionadas en la nueva configuración que no estaban presentes en la anterior (`SELECTIONS`); (2) las características de la configuración anterior que han sido eliminadas en la nueva (`UNSELECTIONS`); y (3) las características que permanecen en la nueva configuración pero cambian sus valores con respecto a la anterior (`MODIFICATIONS`).

5. Propagación de los cambios a la arquitectura

En esta sección definimos el tercer algoritmo de nuestro proceso de evolución, que genera un modelo arquitectónico en CVL para propagar los cambios a la arquitectura desplegada. En primer lugar, con el fin de definir el algoritmo de forma precisa, es necesario formalizar los diferentes modelos de CVL — es decir, el modelo de variabilidad y los modelos de resolución. La formalización de CVL se encuentra parcialmente publicada en [6], pero sólo formaliza la parte abstracta (es decir, el árbol de características) del modelo de variabilidad, y no

los puntos de variación ni los modelos de resolución. Como parte de este trabajo, completamos la especificación definida en [6] para formalizar completamente los modelos de variabilidad y las configuraciones en CVL.

5.1. Formalización de los puntos de variación en CVL

Los puntos de variación (VPs, de *variation points*) definen los elementos del modelo arquitectónico que son variables y pueden ser modificados. Estos también especifican cómo esos elementos variables se modifican mediante transformaciones de modelo (e.g., en ATL [11]). La semántica de estas transformaciones es específica de cada tipo de punto de variación. Por ejemplo, algunos puntos de variación soportados por CVL son la existencia o no de elementos en la arquitectura (`ObjectExistence`), la existencia de relaciones entre los elementos (`LinkExistence`), o la asignación de un valor a una variable (`ParametricSlotAssignment`), entre otros [6].Un tipo importante de punto de variación es `Opaque Variation Point` (`OVP`) que permite definir nuevos puntos de variación personalizados y, por lo tanto, nuevas transformaciones de modelo que no están predefinidas en CVL. Durante la ejecución de CVL, el motor CVL delega su control en un motor de transformaciones modelo a modelo (M2M) encargado de ejecutar las transformaciones definidas por los puntos de variación.

Para representar los puntos de variación, definimos una tupla: $variationPoints = (VP, type, ovptype, semantic, binding, MOFRefs)$, cuyos elementos son:

VP. Conjunto finito, no vacío, de identificadores (nombres únicos) de los puntos de variación.
$type : VP \rightarrow VPType$. Función que dado un punto de variación devuelve su tipo de la taxonomía de puntos de variación disponible en CVL.
$ovpType : VP \rightarrow OVPType$. Función parcial que dado un OVP, devuelve el tipo de ese OVP.
$semantic : OVPType \rightarrow SemanticSpec$. Función que devuelve la semántica de un tipo de OVP. Esto incluye tanto la transformación de modelo a ejecutar por el motor M2M de CVL, como el lenguaje de transformación (e.g., ATL) usado por la transformación.
$binding : VP \rightarrow VSPEC$. Devuelve la característica del árbol asociada al punto de variación.
$refs : VP \rightarrow P(MOFRef)$. Función que devuelve las referencias a elementos de la arquitectura que están enlazadas con el punto de variación. A esos elementos se le aplicarán las transformaciones.

5.2. Formalización de los modelos de resolución en CVL

Dado un modelo de variabilidad V, un modelo de resolución R para V es una colección de características seleccionadas o resueltas ($VSPEC_{res}$) del modelo de variabilidad V. Estas características pueden ser de tres tipos según el tipo de resolución que requieren: (1) $CHOICE_{res}$, aquellas características que se deciden positivamente o negativamente indicando que estarán presentes o no en la configuración; (2) $VARIABLE_{res}$, aquellas características que requieren asignar un valor a una variable; y (3) $CLASSIFIER_{res}$, aquellas características clonables que requieren especificar el número de instancias que se generarán. Cada selección/resolución en R resuelve exactamente una característica de V.

La formalización de un modelo de resolución R es idéntica a la formalización de V, incorporando además las siguientes definiciones:

$VSPEC_{res}$. Colección finita, de identificadores (nombres únicos) de las características (VSpecs) seleccionadas. El conjunto $VSPEC_{res}$ está particionado en $CHOICE_{res}$, $VARIABLE_{res}$, y $CLASSIFIER_{res}$. $CLASSIFIER_{res}$ incluirá todas las instancias de la característica clonable `TenantSpecific`, con un prefijo diferente para cada instancia (e.g., `MálagaTenant:TenantSpecific`). El mismo prefijo es usado para los hijos de esa instancia (e.g., `MálagaTenant:Authentication`).
$resolved : VPSEC_{res} \rightarrow VSPEC$. Función que dada una característica seleccionada en el modelo de configuración (e.g., `MálagaTenant:Authentication`), devuelve la característica original del modelo de variabilidad (e.g., `Authentication`).

$decision : CHOICE_{res} \rightarrow Boolean$. Dada una característica de tipo $CHOICE_{res}$, devuelve verdadero si la característica fue seleccionada en la configuración o falso en caso contrario.
$value : VARIABLE_{res} \rightarrow Value$. Función que dada una característica de tipo $VARIABLE_{res}$, devuelve el valor asignado a ella en la configuración.

5.3. Generación del modelo de composición

El algoritmo para generar el modelo de composición (*Create Weaving Model*) está definido en la Figura 6. El algoritmo recibe como entrada: (1) la nueva configuración evolucionada (obtenida del algoritmo *Evolve Configuration*), (2) las diferencias entre la configuración anterior y la nueva configuración evolucionada (obtenidas del algoritmo *Difference Configuration*), y (3) dos arquitecturas software; la arquitectura software de la aplicación completa ya evolucionada y la arquitectura software correspondiente a la configuración actualmente desplegada. La salida generada es un modelo en CVL con la información de los elementos arquitectónicos que tienen que ser añadidos, eliminados, y/o reconfigurados en la configuración actual de la arquitectura.

El modelo de composición es una extensión del modelo de variabilidad de CVL, que incluye la configuración a desplegar junto con los puntos de variación asociados y las transformaciones de modelo a ejecutar en cada punto de variación por el motor de CVL (`CVL execution engine`). De esta manera, primero el algoritmo extiende el modelo de configuración evolucionado para incluir las diferencias, en forma de nuevas características, en la configuración del modelo de composición (líneas 2 y 3 del algoritmo en la Figura 6). A continuación, el algoritmo genera un punto de variación por cada diferencia en la configuración (líneas 5-32) asociando el tipo apropiado de punto de variación con el tipo de cambio requerido. Por ejemplo, para sustituir un elemento (e.g., un componente) existente en la arquitectura, asociamos un punto de variación del tipo `FragmentSubstitution` (líneas 13-15). Para añadir un nuevo elemento a la arquitectura usamos un `OVP` (*Opaque Variation Point*) con una transformación de modelo encargada de componer (*weave*) el nuevo elemento (líneas 16-21), que puede ser diferente para cada característica (línea 17). Para actualizar el valor de un parámetro usamos el punto de variación `ParametricSlotAssignment` (líneas 22-23). Por último, para eliminar un elemento existente de la arquitectura usamos otro `OVP` con una transformación de modelo que realice la operación de descomposición (*unweaving*) (líneas 25-29).

Finalmente, el modelo de composición es ejecutado por CVL para generar la arquitectura evolucionada con los cambios requeridos para cada tenant.

6. Evaluación

En esta sección evaluamos la eficiencia y corrección de nuestra propuesta calculando la complejidad en tiempo de los algoritmos presentados en las Secciones 4 y 5, y modelamos los modelos de variabilidad y configuraciones como un Problema de Satisfacción de Restricciones (CSP, de *Constraint Satisfaction Problem*). Además, se proporciona una implementación Java de los algoritmos [3].

6.1. Eficiencia de los algoritmos de evolución

Para evaluar la eficiencia de los algoritmos analizaremos la complejidad según el número de operaciones básicas en función del tamaño de la entrada [3].

[3] En `http://150.214.108.91/code/cvl` y `http://150.214.108.91/code/cvltool`.

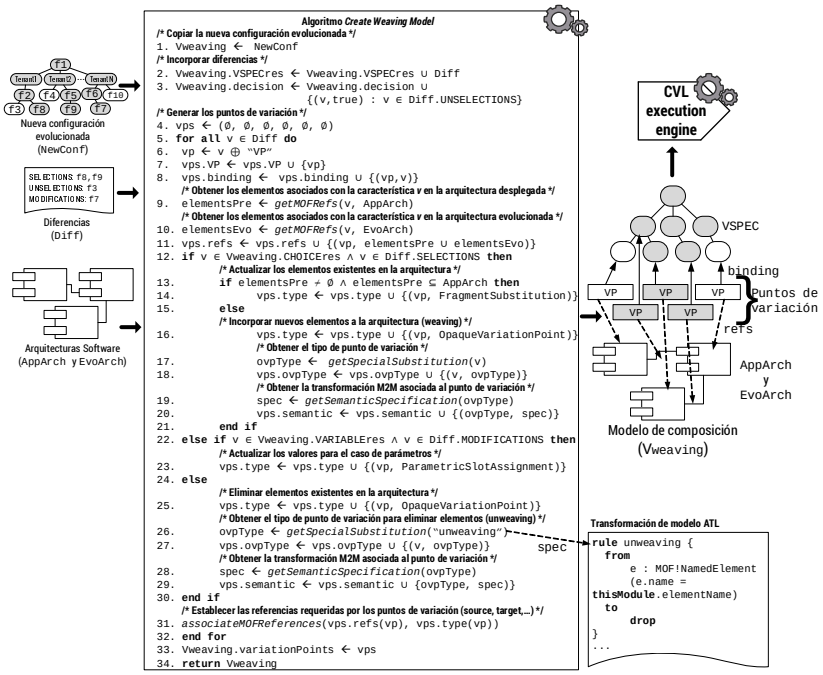

Figura 6. Algoritmo para generar el modelo de composición en CVL.

Consideramos las siguientes operaciones básicas: (i) la unión de conjuntos con un solo elemento que se corresponde con la incorporación de una característica al modelo de variabilidad o configuración; (ii) la diferencia de conjuntos con un elemento que representa la eliminación de una característica del modelo; y (iii) la comprobación de pertenencia de un elemento a un conjunto. Formalmente, sea A el conjunto de características de un modelo y x una característica dada. El tamaño de la entrada de los algoritmos de evolución es el tamaño del modelo de variabilidad (m, el número de características) y el tamaño del modelo de resolución (n, el número de características seleccionadas/resueltas). El tamaño del modelo de resolución depende del número de tenants (t) — es decir, t es el número de instancias de las características clonables en el modelo de configuración. Para simplificar, consideramos $n = m \times t$ como el peor caso, en el cual todas las posibles resoluciones para cada característica clonable han sido seleccionadas. Normalmente $n \leq m \times t$ debido a las restricciones del modelo de variabilidad.

Tabla 1. Complejidad de los algoritmos de evolución.

Algoritmo	Complejidad en tiempo
Evolve Configuration (Figura 3)	$\mathcal{O}(5n^2 + \frac{3}{t}n^2 + (9 + t + \frac{1}{t})n)$
Difference Configuration (Figura 5)	$\mathcal{O}(4n^2 + (8 + t)n)$
Create Weaving Model (Figura 6)	$\mathcal{O}(3n^3 + 18n^2 + 3n)$

Para el algoritmo *Evolve Configuration* el peor caso viene dado para las entradas con una configuración previa de tamaño $n = m \times t$ y unos nuevos requisitos de tamaño también $n = m \times t$, lo que implica cambios en todas las características. La eficiencia del algoritmo puede capturarse fácilmente siguiendo la notación $\mathcal{O}$. Por ejemplo, crear un nuevo modelo de configuración conforme

a la estructura del modelo de variabilidad evolucionado (línea 1 en la Figura 3) conlleva $\mathcal{O}(m{\cdot}n)$ operaciones. Copiar la configuración previa (línea 2) requiere de $\mathcal{O}((3+\frac{1}{t})n^2+t{\cdot}n)$ operaciones en el peor caso. Siguiendo un análisis similar para el resto del algoritmo, éste tiene una complejidad $\mathcal{O}(n^2)$. La Tabla 1 muestra la complejidad computacional para los tres algoritmos. Los dos primeros tienen una complejidad cuadrática ($\mathcal{O}(n^2)$), mientras que la complejidad del tercer algoritmo es cúbica ($\mathcal{O}(n^3)$) en términos de n.

La Figura 7 muestra los resultados de los experimentos empíricos realizados para los tres algoritmos. La eficiencia en los tres casos depende del número de tenants (t) y del número de características del modelo de variabilidad evolucionado (m). Los experimentos se llevaron a cabo en un ordenador portátil Intel Core i3 M350, 2.27GHz, 4 GB de memoria principal, y con la versión 1.7 de la máquina virtual de Java. Los resultados muestran que para evolucionar una configuración con 1000 tenants y 1000 características resueltas para cada tenant, el algoritmo *Evolve Configuration* tarda 50 segundos de media. Para calcular la diferencia entre dos configuraciones, el algoritmo tarda 45 segundos. Finalmente, crear el modelo de composición tarda alrededor de 6 minutos en el peor caso — i.e., cuando todas las características cambian, y por lo tanto, se debe definir un punto de variación para cada una de ellas. En cualquier caso, la eficiencia es aceptable para modelos de variabilidad y configuración de gran tamaño (e.g., un millón de características).

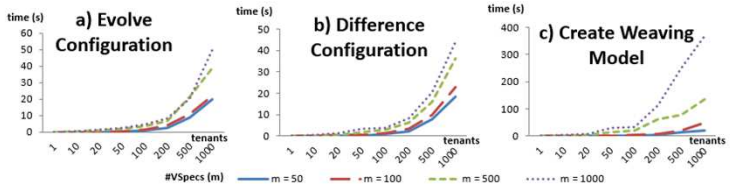

Figura 7. Eficiencia de los algoritmos de evolución.

6.2. Corrección de los algoritmos de evolución

Para demostrar la corrección de los algoritmos de evolución, hemos modelado el modelo de variabilidad y de configuración usando Choco (`http://choco-solver.org/`) una biblioteca Java para Problemas de Satisfacción de Restricciones (CSPs) [16]. Un problema CSP está definido por una tripleta (X, D, C), dónde X es el conjunto de variables, D es el dominio de las variables, y C es el conjunto de restricciones. Mapeamos el modelo de variabilidad en CVL a estos conceptos: (i) las variables son las características del modelo de variabilidad; (ii) el dominio es $\{0, 1\}$ para indicar si la característica ha sido seleccionada o no en una configuración; y (iii) las restricciones, que incluyen las relaciones padre-hijo del árbol y las restricciones OCL. Choco permite generar automáticamente un conjunto mínimo de todas las posibles configuraciones que satisfacen el conjunto inicial de restricciones. Para ello, definimos una función objetivo en CSP ($\sum_{i=1}^{n} v_i$) que minimiza el número de características seleccionadas — i.e., el número de variables con el valor 1. Se puede comprobar que la salida de los algoritmos se corresponde con una de las tuplas $v = \{v_1, \ldots, v_n\}$ generadas por Choco.

7. Trabajo relacionado

A pesar de existir multitud de trabajos centrados en modelar la variabilidad usando características clonables [7,8,17], sólo algunos de ellos tienen en cuenta el problema de la evolución de una familia de productos y los correspondientes cambios en los productos específicos [8,17]. La mayoría de los trabajos gestionan la variabilidad usando modelos de características clásicos (*feature models*) en una SPL [1,5,8], o arquitecturas de referencia [19]. Los modelos de características tienen la ventaja de ser muy conocidos y hay muchas herramientas que le dan soporte (e.g., Hydra Tool presentado en [8]). Sin embargo, el principal inconveniente de los modelos de características clásicos es que requieren un proceso adicional para establecer las relaciones entre la variabilidad especificada a nivel abstracto (e.g., en el árbol) y los puntos de variación en la arquitectura software. En [8], un lenguaje independiente para modelar la variabilidad (VML, de Variability Modeling Language) [13] es usado para establecer la correspondencia entre las características del árbol y las acciones a realizar en la arquitectura software. VML depende del lenguaje usado para modelar la arquitectura, y por lo tanto, es necesario crear manualmente un fichero VML por cada modelo de características y por cada lenguaje de modelado de arquitectura. CVL, por el contrario, facilita la propagación de los cambios a la arquitectura definiendo puntos de variación como parte del modelo de variabilidad, que además soporta cualquier arquitectura definida en lenguajes basados en MOF.

Otro punto a tener en cuenta es que la mayoría de las propuestas existentes se centran en modelar la variabilidad de propiedades no funcionales de las aplicaciones multi-tenant, como por ejemplo el precio de los servicios, su disponibilidad o satisfacción de los usuarios [5,9,14], o se centran en analizar cómo las diferentes variaciones en los servicios afectan a los atributos de calidad de la arquitectura en términos no funcionales (e.g., rendimiento, eficiencia, etc.) [19]. Aunque ambos son aspectos de gran relevancia en el desarrollo de cualquier aplicación software, ninguna de las propuestas existentes aborda el modelado de la variabilidad de los componentes funcionales de la arquitectura (ej: la variabilidad de un componente de autenticación), como proponemos en este artículo.

8. Conclusiones y trabajo futuro

En este artículo hemos presentado una propuesta que usa el lenguaje CVL y los modelos de variabilidad con cardinalidad para gestionar la variabilidad y evolución de un número elevado de tenants en el contexto de aplicaciones en la nube. Evolucionar miles de configuraciones en aplicaciones multi-tenant es una tarea intratable de realizar manualmente. Hemos definido tres algoritmos que automáticamente evolucionan una configuración previa de los tenants, calculan los cambios que se deben hacer en la arquitectura de la aplicación, y propagan dichos cambios a la arquitectura multi-tenant usando transformaciones modelo a modelo. Hemos formalizado los modelos CVL como un problema CSP para demostrar la corrección de los algoritmos y analizar la eficiencia de los algoritmos.

Nuestro trabajo futuro incluye la reconfiguración dinámica de las arquitecturas multi-tenant ejecutando los modelos CVL en tiempo de ejecución.

Agradecimientos

Trabajo financiado por los proyectos MAGIC P12-TIC1814 y HADAS TIN2015-64841-R, y por la Universidad de Málaga.

Referencias

1. Abu Matar, M., Mizouni, R., Alzahmi, S.: Towards software product lines based cloud architectures. In: IEEE IC2E. pp. 117–126 (2014)
2. Armbrust, M., Fox, A., Griffith, R., Joseph, A.D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., Zaharia, M.: A view of cloud computing. Commun. ACM 53(4), 50–58 (2010), http://doi.acm.org/10.1145/1721654.1721672
3. Arora, S., Barak, B.: Computational Complexity: A Modern Approach. Cambridge University Press (2009)
4. Bosch, J.: Design and use of software architectures: adopting and evolving a product-line approach. Pearson Education (2000)
5. Cavalcante, E., Almeida, A., Batista, T., Cacho, N., Lopes, F., Delicato, F.C., Sena, T., Pires, P.F.: Exploiting software product lines to develop cloud computing applications. In: Software Product Line Conference. pp. 179–187. SPLC (2012)
6. CVL Submission Team: Common Variability Language (CVL), OMG revised submission. http://www.omgwiki.org/variability/ (2012)
7. Czarnecki, K., Helsen, S., Eisenecker, U.: Formalizing cardinality-based feature models and their specialization. SP: Improvement and Practice 10(1), 7–29 (2005)
8. Gamez, N., Fuentes, L.: Architectural evolution of famiware using cardinality-based feature models. Information and Software Technology 55(3), 563–580 (2013)
9. García-galán, J., Pasquale, L., Trinidad, P., Ruiz-Cortés, A.: User-centric adaptation analysis of multi-tenant services. ACM Trans. Auton. Adapt. Syst. 10(4), 24:1–24:26 (2016)
10. Haugen, O., Moller-Pedersen, B., Oldevik, J., Olsen, G., Svendsen, A.: Adding standardized variability to domain specific languages. In: SPLC (2008)
11. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: A model transformation tool. Sci. Comput. Program. 72(1–2), 31–39 (2008)
12. Krebs, R., Momm, C., Kounev, S.: Architectural concerns in multi-tenant saas applications. CLOSER 12, 426–431 (2012)
13. Loughran, N., Sánchez, P., Garcia, A., Fuentes, L.: Language support for managing variability in architectural models. In: Software Composition (2008)
14. Mietzner, R., Metzger, A., Leymann, F., Pohl, K.: Variability modeling to support customization and deployment of multi-tenant-aware software as a service applications. In: Principles of Engineering Service Oriented Systems. pp. 18–25 (2009)
15. Pohl, K., Böckle, G., Linden, F.J.v.d.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer-Verlag New York, Inc. (2005)
16. Tsang, E.: Foundations of constraint satisfaction, vol. 289 (1993)
17. White, J., Schmidt, D., Benavides, D., Trinidad, P., Ruiz-Cortes, A.: Automated diagnosis of product-line configuration errors in feature models. In: Software Product Line Conference. pp. 225–234 (2008)
18. Wilder, B.: Cloud Architecture Patterns: Using Microsoft Azure. O'Reilly (2012)
19. Yang, H., Zheng, S., Chu, W.C., Tsai, C.T.: Linking functions and quality attributes for software evolution. In: APSEC. pp. 250–259 (2012)