Automatic Evolution of Eco-Efficient Software Architectures with CVL Models

J. Horcas, and M. Pinto

***Abstract*—Resource sharing and mass storage in server farms provided by cloud platforms save huge amounts of energy. However, optimizing energy consumption at the server room is not enough, being desirable to perform energy optimization of cloud services at the application level. In cloud computing a tailored configuration of services is deployed for each client (tenant), requiring different energy consumption optimizations. Indeed, energy consumption of cloud services depends on several factors determined by the context and usage of the applications. So, to evolve a cloud application to new requirements of energy efficiency implies to perform custom-made adaptations for each tenant. Thus, managing the evolution of a multi-tenant application with hundreds of tenants and thousands of different valid architectural configurations can become intractable if performed manually. This paper proposes a product line architecture approach that: (1) uses cardinality-based variability models to model each tenant as a clonable feature, and (2) automatizes the process of evolving the multi-tenant application architecture when the energy requirements change. The implemented process is efficient for a high number of tenants in a reasonable time.**

***Index Terms*—Cardinality, Energy efficiency, Evolution, Product line architecture, Variability, CVL.**

I. Introducción

UNO de los beneficios de la computación en la nube es que permite ahorrar coste energético, ya que mediante la compartición de recursos y el almacenamiento masivo de datos reduce la necesidad de tener multitud de pequeños centros de datos. Sin embargo, en los últimos años, el número de servicios ofrecidos por las plataformas en la nube (e.g., Microsoft Azure, Amazon Web Services), así como el número de usuarios de estos servicios ha aumentado enormemente, afectando considerablemente a la energía que consume el hardware ofrecido por estas plataformas [1]. De hecho, al integrar los servicios en la nube los desarrolladores están empezando a ser conscientes del gasto energético de estos servicios cuando son utilizados por gran cantidad de usuarios. Sin embargo, reemplazar estos servicios por otros para tener en cuenta nuevas características, como la eficiencia energética, es una tarea complicada. Esta tarea implica modificar la arquitectura software de cada una de las aplicaciones que usan el servicio que queremos reemplazar, quitar o agregar, según un determinado criterio (e.g., la eficiencia energética).

Muchas de las aplicaciones en la nube son desarrolladas siguiendo un enfoque *multitenancy* [2], donde cada variante de una aplicación puede ser personalizada según las necesidades de cada cliente (*tenant*). Gestionar la variabilidad estructural de las aplicaciones *multi-tenant* implica mantener configuraciones de la arquitectura software personalizadas para cada tenant. Las Líneas de Producto Software (SPL, del inglés *Software Product Line*) [3] constituyen un enfoque ampliamente usado para especificar la variabilidad en arquitecturas orientadas a servicios [4], [5]. En el caso de las aplicaciones multi-tenant se plantea un nuevo reto: representar y gestionar de forma explícita configuraciones personalizadas de los mismos servicios, uno para cada tenant [6].

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], [7], la mayoría presenta dos limitaciones: (i) no abordan la automatización de la evolución a nivel de la arquitectura software; y (ii) instancian la SPL de forma individual para cada tenant. Esto aumenta la complejidad de cambiar las configuraciones arquitectónicas de múltiples tenants, dificultándose la realización de los cambios de forma automática y consistente. Además, los estudios más recientes sobre evolución [8], [9] concluyen que los procesos utilizados para soportar la evolución de SPLs son sistemáticos, pero con un bajo nivel de automatización. Esto es debido a la falta de una formalización de los modelos y artefactos usados en las SPLs [9].

Para abordar todos estos retos, este artículo plantea los siguientes objetivos: (1) identificar los cambios necesarios cuando los requisitos de eficiencia energética cambian, tanto de la propia aplicación como de los servicios proporcionados por la plataforma en la nube; y (2) obtener automáticamente, para cada tenant, una arquitectura software evolucionada que sea energéticamente eficiente y que sea válida y consistente con la configuración actualmente desplegada en ese tenant. Para conseguir estos objetivos, en este artículo:

- Se define un proceso para evolucionar una línea de productos arquitectónica (PLA, del inglés *Product Line Architecture*) [2].
- Se modela la configuración de cada tenant como una característica clonable usando modelos de variabilidad con cardinalidad en CVL [10].
- Se formalizan los modelos de configuración y puntos de variación del lenguaje CVL.
- Se automatiza 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.
- Se evalúa la complejidad y eficiencia de los algoritmos para un número elevado de tenants, y eficiencia energética de la arquitectura evolucionada.

El artículo se organiza de la siguiente manera. La Sección

José Miguel Horcas, Universidad de Málaga, Spain, horcas@lcc.uma.es.
Mónica Pinto, Universidad de Málaga, Spain, pinto@lcc.uma.es.

Trabajo financiado por los proyectos MAGIC P12-TIC1814, HADAS TIN2015-64841-R (fondos FEDER), MEDEA RTI2018-099213-B-I00 (fondos FEDER) y TASOVA MCIU-AEI TIN2017-90644-REDT.

II describe los retos principales de la propuesta. La Sección III introduce los conceptos básicos de CVL. La Sección IV explica cómo modelar la variabilidad de aplicaciones multi-tenant con CVL. La Sección V y VI detallan el proceso de evolución. La Sección VII evalúa la propuesta. La Sección VIII y IX discuten el trabajo relacionado y las conclusiones.

II. Motivación y Caso de Estudio

Nuestro caso de estudio es una aplicación para la gestión de servicios en las administraciones públicas. Con el fin de ahorrar costes de implementación y mantenimiento, se decide desarrollar la aplicación usando una plataforma en la nube (e.g., Microsoft Azure) [11]. El objetivo es vender la aplicación personalizada para cada cliente (administraciones públicas), por lo que la aplicación seguirá un enfoque multi-tenant, donde cada administración pública será un tenant configurado a medida. Por lo tanto, la aplicación multi-tenant tendrá un conjunto de servicios comunes a todos los tenants, y también un conjunto de servicios específicos para cada tenant.

Algunos de los servicios comunes son propios de la aplicación administrativa, como un sistema de citas y el historial de trámites de los usuarios. Otros son servicios ofrecidos por la plataforma Azure, como un servicio de persistencia, o la posibilidad de crear múltiples copias de la información en diferentes centros de datos. Por otro lado, los servicios específicos se configuran a medida de cada tenant. Por ejemplo, Madrid es la única provincia que proporciona un servicio de alquiler de inmuebles en línea, por lo que el componente con esta funcionalidad estará incluido únicamente en este tenant. Finalmente, algunos servicios de la plataforma en la nube también pueden configurarse para cada tenant. Por ejemplo, el método de autenticación es diferente en cada administración: la de Málaga usa un certificado digital para autenticar al personal administrativo y un sistema de nombre de usuario y contraseña para los usuarios externos; mientras que la de Sevilla usa autenticación por nombre de usuario y contraseña para todos, y la de Madrid usa certificado digital para todos. Por tanto, 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 en cada tenant se despliega sólo un subconjunto de las características variables. Dado que se deben generar y mantener cientos de configuraciones diferentes de la aplicación, lo que implica gestionar miles de componentes, un reto importante es proporcionar mecanismos para representar y gestionar directamente la variabilidad de las aplicaciones multi-tenant [7].

Pero una vez desplegadas, estas aplicaciones tienen que ser adaptadas a mejoras tecnológicas y a cambios en las necesidades de los clientes. Por ejemplo, aparecen con frecuencia nuevos métodos de autenticación con distinto consumo energético (e.g., autenticación biométrica, autenticación usando redes sociales) o persistencia (e.g., fragmentación de base de datos, grupos de afinidad). Si se quiere proporcionar estos nuevos servicios a los clientes, éstos deben incorporarse a los tenants que lo requieran. También, durante la vida de la aplicación, los clientes pueden exigir nuevas funcionalidades (e.g., un nuevo módulo para la gestión de delitos telemáticos).

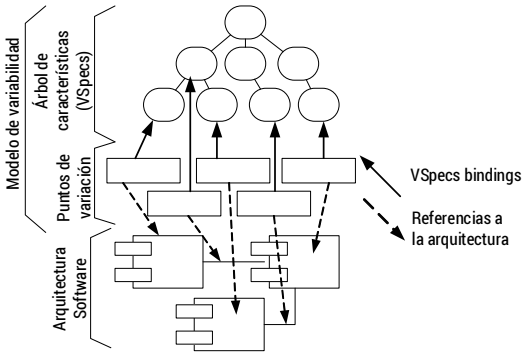

Fig. 1. Esquema del enfoque CVL.

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 valores para sus 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 son el cálculo automático de los cambios a realizar en cada tenant y la propagación automática de estos cambios para todos los tenants a nivel arquitectónico. Por otra parte, este proceso de evolución automático sólo será útil si es eficiente para un gran número de tenants, por lo que es importante evaluar la eficiencia del proceso de evolución.

III. Conceptos Previos: El Lenguaje CVL

CVL (*Common Variability Language*) [10] es un lenguaje independiente del dominio para especificar y resolver la variabilidad en modelos basados en MOF (*MetaObject Facility*).

El modelo de variabilidad CVL está formado por dos partes (Figura 1). 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 1 y especifica la funcionalidad de la aplicación. La segunda parte del modelo de variabilidad son los puntos de variación (*variation points*), que aparecen en la parte central de la Figura 1. 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. 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, resolviendo

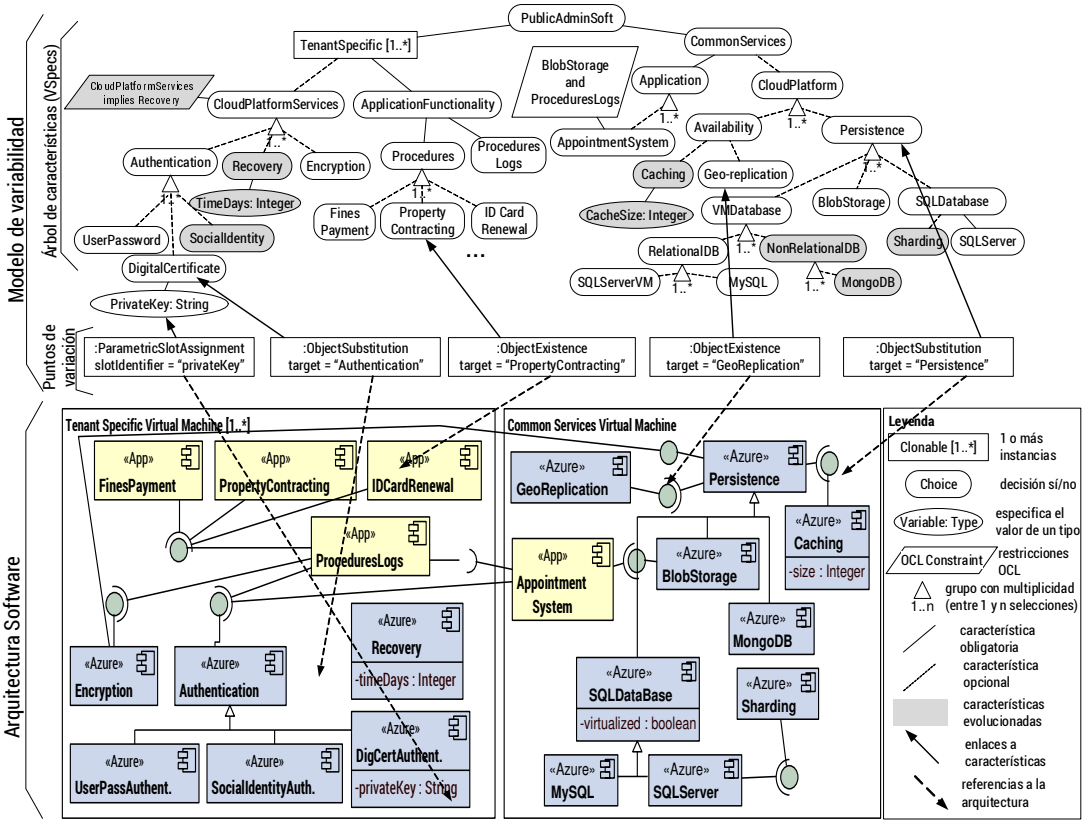

Fig. 2. Modelo de variabilidad en CVL y arquitectura software.

la variabilidad sobre el modelo de la arquitectura software. En CVL una configuración del modelo de variabilidad recibe el nombre de modelo de resolución (*resolution model*).

Las siguientes secciones presentan nuestra propuesta para modelar la variabilidad y gestionar la evolución de las aplicaciones multi-tenant con CVL.

IV. Gestión de la variabilidad con CVL

Para dar soporte a diferentes configuraciones para cada tenant, nuestra propuesta define la funcionalidad específica de los tenant bajo una característica clonable [6]. Como muestra el árbol de la Figura 2, se modela explícitamente la variabilidad de las características que son específicas de cada tenant (i.e., sub-árbol `TenantSpecific[1..*]`) y también la funcionalidad común que comparten todos los tenants (i.e., sub-árbol `CommonServices`). La característica clonable tiene una cardinalidad `[1..*]` que indica que puede ser instanciada una o más veces y todas sus sub-características pueden ser configuradas de forma diferente. En nuestro ejemplo, la cardinalidad representa el número de tenants, y cada instancia de `TenantSpecific[1..*]` define la configuración para ese tenant específico (e.g., para la administración de Málaga). Seleccionando las características apropiadas bajo la característica clonable `TenantSpecific[1..*]`, y ejecutando CVL, nuestra propuesta genera una configuración de la arquitectura donde cada tenant está configurado de acuerdo a sus necesidades.

V. Gestión de la Evolución con CVL

Una vez que se ha generado y desplegado una configuración arquitectónica a medida de cada tenant, la aplicación multi-tenant es susceptible de evolucionar debido a mejoras tecnológicas (como componentes que consumen menos energía) y/o a cambios en las necesidades de los clientes. Volviendo a nuestro caso de estudio, supongamos que Microsoft incorpora nuevas funcionalidades a su plataforma que consumen menos energía: un servicio de recuperación de datos, un método de autenticación basado en Facebook y un mecanismo de persistencia. Además, el proveedor de la aplicación administrativa quiere proporcionar estos nuevos servicios a sus tenants.

La Figura 3 muestra el esquema con los diferentes pasos en el proceso de evolución. 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 para nuestro caso de estudio). Por ejemplo, las características `Recovery`, `SocialIdentity`, `Caching`, `MongoDB` y `Sharding` han sido incorporadas en el modelo de variabilidad con el fin de añadir los nuevos servicios de recuperación, de autenticación, y 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. El objetivo es que esas

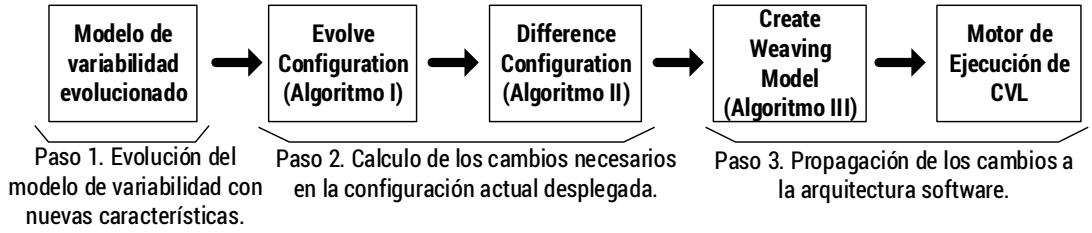

Fig. 3. Gestión de la evolución con CVL.

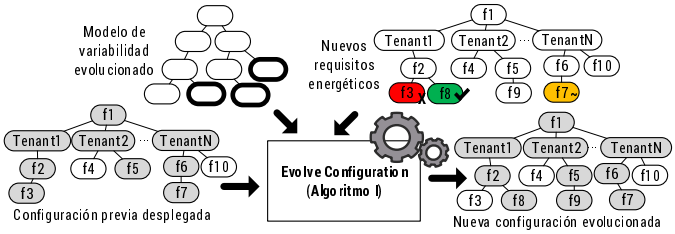

Fig. 4. Algoritmo I.

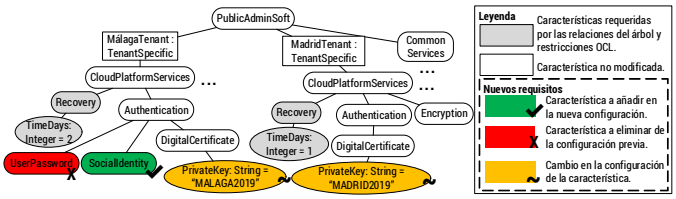

Fig. 5. Evolución de una configuración específica.

configuraciones ya desplegadas satisfagan el nuevo modelo de variabilidad y los nuevos requisitos de la aplicación. Como muestra la Figura 3, 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 1: *Evolve Configuration*); y (ii) calcular las diferencias entre la configuración evolucionada y la configuración actualmente desplegada (Algoritmo 2: *Difference Configuration*).

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. Siguiendo con la Figura 3, se define 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 [12]) sólo abordan el problema de la evolución a nivel de características, y requieren modificar manualmente la arquitectura software para reflejar los cambios calculados (e.g., modificar la configuración de 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.

El Algoritmo I (Figura 4) para evolucionar la configuración de una aplicación multi-tenant recibe como entrada el modelo de la configuración actualmente desplegada, el modelo de variabilidad 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 con todas las configuraciones de los tenants evolucionados.

La Figura 5 muestra una vista parcial del árbol de características donde se ha representado en el mismo árbol las tres entradas del algoritmo. El algoritmo genera un nuevo

Algoritmo 1 Evolve Configuration.

Inputs: V: Modelo de variabilidad evolucionado.
R_{pre}: Configuración previa.
Req: Nuevos requisitos.
Output: $(R_{new}, Boolean)$: Nueva configuración evolucionada.
1: $R_{new} \leftarrow V$ /* La configuración respeta el modelo de variabilidad */
/* Copia configuraciones previa */
2: $R_{new}.VSPEC_{res} \leftarrow \{(v \in R_{pre}.VSPEC_{res} : R_{pre}.resolved(v) \in V.VSPEC\}$
3: $R_{new}.resolved \leftarrow \{(r, v) \in R_{pre}.resolved : r \in R_{new}.VSPEC_{res}\}$
4: $R_{new}.decision \leftarrow \{(r, x) \in R_{pre}.decision : r \in R_{new}.VSPEC_{res}\}$
5: $R_{new}.value \leftarrow \{(r, x) \in R_{pre}.value : r \in R_{new}.VSPEC_{res}\}$
6: $R_{new}.instance \leftarrow \{(r, v) \in R_{pre}.instance : r \in R_{new}.VSPEC_{res}\}$
/* Añade nuevos requisitos */
7: $R_{new}.VSPEC_{res} \leftarrow R_{new}.VSPEC_{res} \cup Req.VSPEC_{res}$
8: $R_{new}.resolved \leftarrow R_{new}.resolved \cup Req.resolved$
9: $R_{new}.decision \leftarrow (R_{new}.decision \setminus \{(r, x) \in Req.decision : r \in R_{new}.VSPEC_{res}\}) \cup Req.decision$
10: $R_{new}.value \leftarrow (R_{new}.value \setminus \{(r, x) \in Req.value : r \in R_{new}.VSPEC_{res}\}) \cup Req.value$
11: $R_{new}.instance \leftarrow R_{new}.instance \cup Req.instance$
/* Añade características obligatorias */
12: $R_{new}.decision \leftarrow (R_{new}.decision \setminus \{(r, x) \in Req.decision : V.mandatory(Req.resolved(r))\}) \cup \{(v, true) \in V.mandatory\}$
/* Comprueba si todas las configuraciones están presentes */
13: $full \leftarrow$ fullConfigurationModel(R_{new}, V)
14: **return** $(R_{new}, full)$

modelo donde, en primer lugar, se copian aquellas características que no cambian (características en color blanco). A continuación, se añaden las nuevas características seleccionadas (marcadas con ✔ en color verde), se omiten aquellas características que ya no son requeridas (marcadas con ✗ en color rojo) y se añaden las características cuyos valores han cambiado (marcadas con ∼ en color amarillo). Finalmente, se añaden aquellas características 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 (``MALAGA2019'') como para el tenant Madrid (``MADRID2019''). 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`

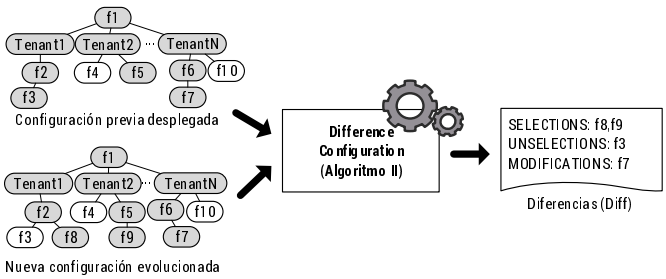

Fig. 6. Algoritmo II.

`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. El algoritmo *Difference Configuration* recibe como entrada dos configuraciones (la configuración previa y la configuración nueva obtenida con el algoritmo *Evolve Configuration*) y calcula la diferencia entre ellas (Figura 6). Las diferencias están determinadas por: (1) las nuevas características seleccionadas en la nueva configuración que no estaban 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`).

Algoritmo 2 Difference Configuration.

Inputs: R_{pre}: Configuración previa.
R_{new}: Nueva configuración evolucionada.
Output: $Diff$: Diferencias.
1: $Diff \leftarrow R_{new}$ /* Configuración bien formada */
2: $CHANGES \leftarrow Diff.VSPEC_{res} \cup R_{pre}.VSPEC_{res}$
3: $CHANGES \leftarrow CHANGES \setminus \{r \in R_{pre}.CHOICE_{res} : R_{pre}.decision(r) = R_{new}.decision(r)\}$ /* Cambios en selecciones */
4: $CHANGES \leftarrow CHANGES \setminus \{r \in R_{pre}.VARIABLE_{res} : R_{pre}.value(r) = R_{new}.value(r)\}$ /* Cambios en variables */
5: $Diff.decision \leftarrow Diff.decision \setminus \{(r,x) \in Diff.decision : r \notin CHANGES\}$
6: $Diff.variable \leftarrow Diff.variable \setminus \{(r,x) \in Diff.variable : r \notin CHANGES\}$
7: $Diff.instance \leftarrow Diff.instance \setminus \{(r,v) \in Diff.instance : r \notin CHANGES\}$
8: $Diff.VSPEC_{res} \leftarrow CHANGES$
9: **return** $Diff$

VI. Propagación de Cambios a la Arquitectura

Esta sección presenta 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 [13], 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, se completa la especificación definida en [13] para formalizar completamente los modelos de variabilidad y las configuraciones en CVL.

A. 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 [14]). La semántica de estas transformaciones es específica de cada punto de variación. Por ejemplo, algunos puntos de variación soportados por CVL son la existencia 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. Otro punto de variación importante es `Opaque Variation Point` (`OVP`) que permite definir nuevos puntos de variación personalizados junto con nuevas transformaciones de modelo no definidas en CVL. Durante la ejecución de CVL, el motor CVL delega su control en un motor de transformaciones 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`.** 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 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.

B. 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 (`VSPECres`) del modelo de variabilidad V. Estas características pueden ser de tres tipos según el tipo de resolución que requieren: (1) `CHOICEres`, aquellas características que se deciden positivamente o negativamente indicando que estarán presentes o no en la configuración; (2) `VARIABLEres`, aquellas características que requieren asignar un valor a una variable; y (3) `CLASSIFIERres`, 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:

- **`VSPECres`.** Colección finita, de identificadores (nombres únicos) de las características (*VSpecs*) seleccionadas. El conjunto `VSPECres` está particionado en `CHOICEres`, `VARIABLEres`, y `CLASSIFIERres`. `CLASSIFIERres` 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: VPSECres` → `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: CHOICEres` → `Boolean`.** Dada una característica de tipo `CHOICEres`, devuelve verdadero si la característica fue seleccionada en la configuración.
- **`value: VARIABLEres` → `Value`.** Función que dada una característica de tipo `VARIABLEres`, devuelve el valor asignado a ella en la configuración.

C. Generación del Modelo de Composición

El algoritmo para generar el modelo de composición (Algoritmo 3: *Create Weaving Model*) 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*), (3) la arquitectura software de la aplicación ya evolucionada y, (4) 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 (Figura 7).

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 III). 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 se usa 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 use usa el punto de variación `ParametricSlotAssignment` (líneas 22-23). Por último, para eliminar un elemento existente de la arquitectura se usa 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.

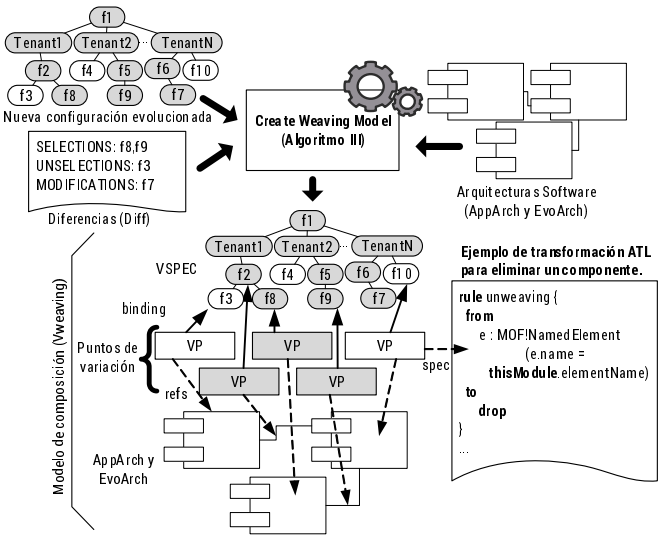

Fig. 7. Algoritmo III.

Algoritmo 3 Create Weaving Model.

Inputs: $NewConf$: Nueva configuración evolucionada.
$Diff$: Diferencias.
$EvoArch$: Arquitectura software evolucionada.
$AppArch$: Arquitectura previa desplegada.
Output: $V_{weaving}$: Modelo de Weaving.

1: $V_{weaving} \leftarrow NewConf$ /* Modelo bien formado */
/* Incorporar diferencias */
2: $V_{weaving}.VSPEC_{res} \leftarrow V_{weaving}.VSPEC_{res} \cup Diff$
3: $V_{weaving}.decision \leftarrow V_{weaving}.decision \cup \{(v, true) : v \in Diff.UNSELECTIONS\}$
4: $vps \leftarrow (\varnothing, \varnothing, \varnothing, \varnothing, \varnothing, \varnothing)$
5: **for all** $v \in Diff$ **do**
6: $vp \leftarrow v \oplus \texttt{VP}$ /* Concatenación de cadenas */
7: $vps.VP \leftarrow vps.VP \cup \{vp\}$
8: $vps.binding \leftarrow vps.binding \cup \{(vp, v)\}$
9: $elements_{pre} \leftarrow$ getMOFRefs(v, $AppArch$) /* Componentes asociados a la característica v en la configuración previa */
10: $elements_{evo} \leftarrow$ getMOFRefs(v, $EvoArch$) /* Componentes asociados a la característica v en la configuración evolucionada */
11: $vps.refs \leftarrow vps.refs \cup \{(vp, elements_{pre} \cup elements_{evo})\}$
12: **if** $v \in V_{weaving}.CHOICE_{res} \wedge Diff.SELECTIONS$ **then**
13: **if** $elements_{pre} \neq \varnothing \wedge elements_{pre} \subseteq M_{App}$ **then** /* Actualizar componentes */
14: $vps.type \leftarrow vps.type \cup \{(vp, \texttt{FragmentSubstitution})\}$
15: **else** /* Incoporar nuevos componentes */
16: $vps.type \leftarrow vps.type \cup \{(vp, \texttt{OpaqueVariationPoint})\}$
17: $ovpType \leftarrow$ getSpecialSubstitution(v) /* Patrón de weaving */
18: $vps.ovpType \leftarrow vps.ovpType \cup \{(v, ovpType)\}$
19: $spec \leftarrow$ getSemanticSpec($ovpType$) /* Transformación M2M */
20: $vps.semantic \leftarrow vps.semantic \cup \{(ovpType, spec)\}$
21: **end if**
22: **else if** $v \in V_{weaving}.VARIABLE_{res} \wedge v \in Diff.MODIFICATIONS$ **then** /* Actualizar variables */
23: $vps.type \leftarrow vps.type \cup \{(vp, \texttt{ParametricSlotAssignment})\}$
24: **else** /* Eliminar componentes */
25: $vps.type \leftarrow vps.type \cup \{(vp, \texttt{OpaqueVariationPoint})\}$
26: $ovpType \leftarrow$ getSpecialSubstitution(`unweaving`) /* Patrón unweaving */
27: $vps.ovpType \leftarrow vps.ovpType \cup \{(v, ovpType)\}$
28: $spec \leftarrow$ getSemanticSpec($ovpType$) /* Transformación M2M */
29: **end if**
30: associateMOFReferences($vps.refs(vp)$, $vps.type(vp)$) /* Actualiza origen y destino de las referencias de acuerdo al tipo de punto de variación. */
31: **end for**
32: $V_{weaving}.variationPoints \leftarrow vps$
33: **return** $V_{weaving}$

VII. Evaluación

Esta sección evalúa, en términos de rendimiento energético, los beneficios de aplicar nuestra propuesta en aplicaciones multi-tenant. Se evalúa también la eficiencia de nuestra propuesta calculando la complejidad de los algoritmos.

A. Rendimiento Energético en Aplicaciones Multi-Tenant

Para discutir los beneficios en eficiencia energética que se obtienen al aplicar nuestra propuesta se plantea un nuevo escenario [11][1]: supongamos que la aplicación multi-tenant ya dispone de un servicio de grabación y almacenamiento remoto de las llamadas de los usuarios a las administraciones. Este servicio graba las llamadas de voz y las envía cada cierto tiempo a un servidor remoto para su almacenamiento. Cada administración puede configurar la frecuencia con la que se almacenan los datos en el servidor remoto. Hasta ahora la información se enviaba sin comprimir, pero se desea evolucionar la aplicación para incluir un servicio de compresión que comprima los datos antes de mandarlos al servidor. Como el patrón de uso de cada tenant es diferente, cada tenant podrá elegir el algoritmo de compresión que considere más apropiado utilizar. Por ejemplo, la administración de Madrid recibe un volumen de llamadas mucho mayor que la administración de Málaga, por lo que decide usar un algoritmo con un ratio de compresión mayor (e.g., LAME); mientras que la administración de Málaga escoge el algoritmo VORBIS.

Aplicando nuestra propuesta, la aplicación multi-tenant es evolucionada de forma que a la configuración de cada tenant se añada el nuevo servicio de compresión, configurado a medida de cada tenant. La Figura 8 muestra la diferencia en consumo energético entre la configuración anterior (configuración presentada en la Sección II), en la que no se usaba el servicio de compresión ("Sin compresión"), y las diferentes configuraciones evolucionadas con el servicio de compresión, pero usando diferentes algoritmos. Por ejemplo, la nueva configuración de la administración de Madrid usando el algoritmo LAME de compresión proporciona un ahorro energético del 88.72% para envíos de 128 MB (613 J) con respecto a la configuración previa (5425 J), mientras que Málaga usando el algoritmo VORBIS presenta un ahorro del 90.67% para envíos de 128 MB (506 J).

B. Eficiencia de los Algoritmos de Evolución

Para evaluar la eficiencia de los algoritmos se analiza la complejidad según el número de operaciones básicas en función del tamaño de la entrada [15]. Se consideran 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

[1]Los resultados completos de la evaluación se encuentran en: https://github.com/jmhorcas/IEEE-energy-evaluation

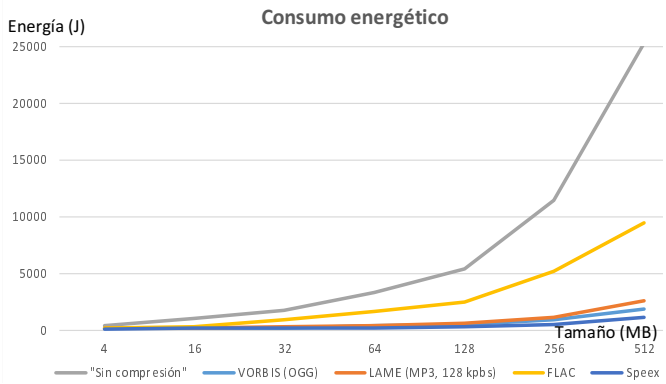

Fig. 8. Consumo energético de algoritmos de compresión.

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). 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, se considera $n = m \cdot t$ como el peor caso, en el cual todas las posibles resoluciones para cada característica clonable han sido seleccionadas. Normalmente $n \leq m \cdot t$ debido a las restricciones del modelo de variabilidad.

La Figura 9 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 Java. Los resultados muestran un incremento cuadrático del tiempo de ejecución para los dos primeros algoritmos, mientras que el crecimiento es cúbico para el tercer algoritmo. En cualquier caso, la eficiencia es aceptable para modelos de variabilidad y configuración de gran tamaño. Por ejemplo, para evolucionar una configuración con 1000 tenants y 1000 características resueltas para cada tenant (i.e., un millón de características en total), el algoritmo *Evolve Configuration* tarda 50 segundos de media. Para calcular la diferencia entre dos configuraciones, el algoritmo *Difference Configuration* tarda 45 segundos. Finalmente, crear el modelo de composición (algoritmo *Create Weaving Model*) tarda alrededor de 6 minutos en el peor caso (i.e., evolucionan todas las características). La diferencia en eficiencia con los otros algoritmos se debe a que el Algoritmo III genera un nuevo modelo CVL con la información obtenida del algoritmo I y II; y además define un punto de variación con su transformación M2M asociada para cada característica que evoluciona.

VIII. Trabajo Relacionado

A pesar de existir multitud de trabajos centrados en modelar la variabilidad usando características clonables [6], [5], [12], solo algunos de ellos tienen en cuenta el problema de la evolución de los productos específicos de una familia de productos [8]. La mayoría de los trabajos gestionan la variabilidad usando modelos de características clásicos (*feature models*) en una SPL [12], o arquitecturas de referencia [16]. Los modelos

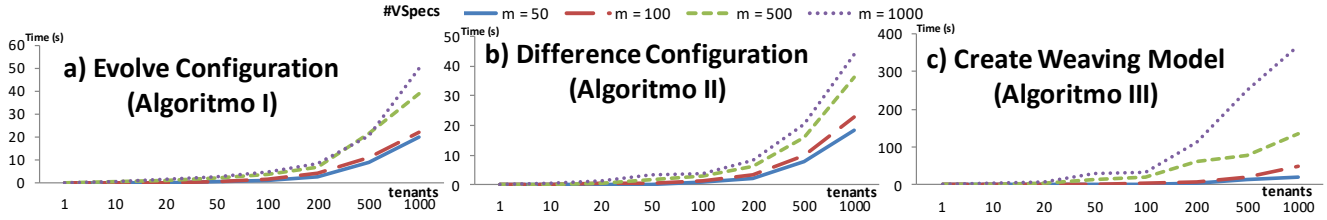

Fig. 9. Eficiencia de los algoritmos de evolución.

de características tienen la ventaja de ser muy conocidos y hay muchas herramientas que le dan soporte [5]. Sin embargo, su principal inconveniente 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 [12], un lenguaje independiente para modelar la variabilidad (VML, de *Variability Modeling Language*) [17] 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 lenguaje de modelado de arquitectura. Por el contrario, el uso de CVL en este artículo supone una gran ventaja respecto a los trabajos que usan *feature models* [3], [12], ya que CVL permite conectar las características del árbol con los elementos de la arquitectura de forma directa mediante los puntos de variación que forman parte del modelo de variabilidad. Esto facilita la propagación de los cambios a la arquitectura, además de soportar cualquier arquitectura basada en MOF.

Por otro lado, los estudios más recientes sobre evolución [8], [9] muestran que no hay consenso sobre la formalización de las SPLs y sobre qué modelos y artefactos pueden evolucionar, ni cómo y cuándo evolucionan. Este artículo supone un primer paso para mejorar el estado del arte en la comunidad de SPL, proporcionando una formalización de los modelos involucrados en la evolución así como de los algoritmos para evolucionar dichos modelos de forma automática.

IX. Conclusiones

Evolucionar miles de configuraciones en aplicaciones multi-tenant cuando los requisitos de eficiencia energética cambian es una tarea intratable de realizar manualmente. Este artículo presenta un proceso de evolución 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. El artículo proporciona una formalización de los modelos de una SPLs, así como los algoritmos para evolucionar dichos modelos. Gracias a estos algoritmos de evolución, los cambios tecnológicos y los cambios en los requisitos de las aplicaciones se propagan automáticamente a las arquitecturas previamente desplegadas.

Referencias

[1] L. F. Grisales, O. D. Montoya, A. Grajales, R. A. Hincapie, and M. Granada, "Optimal planning and operation of distribution systems considering distributed energy resources and automatic reclosers," *IEEE Latin America Transactions*, vol. 16, no. 1, pp. 126–134, Jan 2018.

[2] R. Krebs, C. Momm, and S. Kounev, "Architectural concerns in multi-tenant saas applications." *Closer*, vol. 12, pp. 426–431, 2012.

[3] J. M. Horcas, M. Pinto, and L. Fuentes, "An automatic process for weaving functional quality attributes using a software product line approach," *Journal of Systems and Software*, vol. 112, pp. 78–95, 2016.

[4] ——, "Variability models for generating efficient configurations of functional quality attributes," *Information & Software Technology*, vol. 95, pp. 147–164, 2018.

[5] M. Raatikainen, J. Tiihonen, and T. Männistö, "Software product lines and variability modeling: A tertiary study," *Journal of Systems and Software*, vol. 149, pp. 485–510, 2019.

[6] K. Czarnecki, S. Helsen, and U. W. Eisenecker, "Formalizing cardinality-based feature models and their specialization," *Software Process: Improvement and Practice*, vol. 10, no. 1, pp. 7–29, 2005.

[7] Y. Fernandez Perez, C. Cruz Corona, and J. L. Verdegay Galdeano, "A new model based on soft computing for evaluation and selection of software products," *IEEE Latin America Transactions*, vol. 16, no. 4, pp. 1186–1192, April 2018.

[8] J. L. Barros Justo, N. Martinez Araujo, and A. Gonzalez Garcia, "Software reuse and continuous software development: A systematic mapping study," *IEEE Latin America Transactions*, vol. 16, no. 5, pp. 1539–1546, May 2018.

[9] M. Marques, J. Simmonds, P. O. Rossel, and M. C. Bastarrica, "Software product line evolution: A systematic literature review," *Information and Software Technology*, vol. 105, pp. 190–208, 2019.

[10] J. M. Horcas, M. Pinto, and L. Fuentes, "Extending the common variability language (CVL) engine: A practical tool," in *Systems and Software Product Line Conference*, ser. SPLC, 2017, pp. 32–37.

[11] C. M. S. Ferreira, M. J. P. Peixoto, P. A. S. Duarte, A. B. B. Torres, M. L. Silva Junior, L. S. Rocha, and W. Viana, "An evaluation of cross-platform frameworks for multimedia mobile applications development," *IEEE Latin America Transactions*, vol. 16, no. 4, pp. 1206–1212, April 2018.

[12] N. Gamez and L. Fuentes, "Architectural evolution of famiware using cardinality-based feature models," *Information and Software Technology*, vol. 55, no. 3, pp. 563–580, 2013.

[13] CVL Submission Team, "Common Variability Language (CVL), OMG revised submission," http://www.omgwiki.org/variability/, 2012.

[14] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev, "ATL: A model transformation tool," *Sci. Comp. Prog.*, vol. 72, pp. 31–39, 2008.

[15] S. Arora and B. Barak, *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009.

[16] H. Yang, S. Zheng, W.-C. Chu, and C.-T. Tsai, "Linking functions and quality attributes for software evolution," in *APSEC*, 2012, pp. 250–259.

[17] N. Loughran, P. Sánchez, A. Garcia, and L. Fuentes, "Language support for managing variability in architectural models," in *Soft. Comp.*, 2008.

José Miguel Horcas José Miguel Horcas es investigador en el Departamento de Lenguajes y Ciencias de la Computación de la Universidad de Málaga (España). Recibió su título de Doctor Ingeniero en Informática en 2018. Actualmente, trabaja en Desarrollo de Software Orientado a Aspectos, Líneas de Productos Software y Atributos de Calidad, y ha participado en varios proyectos de investigación nacionales e internacionales.

Mónica Pinto Mónica Pinto es titular de universidad en el Departamento de Lenguajes y Ciencias de la Computación de la Universidad de Málaga (España). Actualmente pertenece al grupo de investigación CAOSD dentro del grupo GISUM. Sus áreas de investigación principales son la Ingeniería del Software basada en Componentes y Aspectos y las Líneas de Productos Software, principalmente para el desarrollo de aplicaciones (móviles) sensibles al contexto y energéticamente eficientes.