Product Line Architecture for Automatic Evolution of Multi-Tenant Applications

Jose-Miguel Horcas, Mónica Pinto, and Lidia Fuentes
Universidad de Málaga, Andalucía Tech, Spain
Email: {horcas,pinto,lff}@lcc.uma.es

***Abstract*—Cloud computing is becoming the predominant mechanism to seamlessly deploy applications with special requirements such as massive storage sharing or load balancing, usually provided as services by cloud platforms. A developer can improve the application's delivery and productivity by following a multi tenancy approach, where variants of the same application can be quickly customized to the necessities of each tenant. However, managing the inherent variability existing in multi-tenant applications and, even more importantly, 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. In this paper we propose a product line architecture approach in which: (1) we use cardinality-based variability models to model each tenant as a clonable feature, (2) we automate the process of evolving the multi-tenant application architecture, and (3) we demonstrate that the implemented process is correct and efficient for a high number of tenants in a reasonable time. We use a running case study in the domain of medical software.**

I. Introduction

Cloud computing [1] is becoming the predominant mechanism to seamlessly deploy applications with special requirements, such as massive storage sharing, automatic scaling, business analytics, and identity secure messaging. These are turnkey services usually provided by cloud platforms[1] (e.g., Microsoft Azure, Amazon Web Services,...) that allow developers to get their applications up and running quickly. Then, application developers can integrate cloud platform services as part of the software architecture of their applications.

Application providers can improve their application delivery processes and productivity even more by following a multi tenancy approach [2], where variants of the same application can be quickly customized to the necessities of each customer. However, managing the inherent variability existing in multi-tenant applications, where the developers need to maintain different configurations of the application software architecture for each tenant, is not a straightforward task. For this reason, the Software Product Line (SPL) [3] paradigm is a widely accepted approach [4], [5], [6] to specify variabilities and commonalities in general, and specifically in service-oriented architectures. However, multi-tenancy poses a new challenge, that is, to explicitly represent and manage the existence of simultaneous configurations of the same services, one for each tenant [7].

Even more important than managing the variability of multi-tenant applications is managing the evolution of these applications. The cloud-platform providers are continuously evolving and updating their technologies in order to be competitive in the market. Moreover, the application specific functionality can evolve to consider new functionality required by the tenants. In both cases, the software architecture of the cloud application needs to be adapted to add new components or to remove and configure the existing ones. However, managing the evolution of a multi-tenant application with hundreds of tenants and thousands of different valid configurations can become intractable if performed manually. Although SPL approaches in the context of multi-tenant applications already exist [4], [8], [9], [5], [6], [10], most of them present two main shortcomings: (i) they do not focus on automating the evolution at the architectural level; (ii) and they apply the SPL approach individually for each tenant. This makes it more difficult to perform the changes automatically and consistently, and increases the complexity of changing the existing configurations of the software architecture of all tenants at the same time.

Thus, the main goal of this paper is twofold: (1) to elicit the changes that need to be performed within the cloud-based applications when the platform's service requirements and/or the application requirements evolve; and (2) to obtain a valid, evolved software architecture for each tenant consistent with the configuration deployed for every tenant. In order to achieve these goals we propose a Product Line Architecture (PLA) [11]-based approach in which: (i) we use cardinality-based variability models of the Common Variability Language (CVL) [12] to model each tenant as a clonable feature, (ii) we automate the process of evolving the multi-tenant architecture by defining three algorithms that automatically propagate the required changes to the architecture configuration of each tenant, and (iii) we demonstrate that the implemented algorithms are correct and efficient enough to be applied to a high number of tenants in a reasonable time. We describe our approach using a running case study from the domain of medical software solutions.

This paper is organized as follows. Section II presents a motivational example and identifies the list of challenges. Section III explains how to model the variability of multi-tenant applications. Section IV and Section V detail our automatic evolution process. In Section VI our approach is validated. Section VII discusses the related work. Finally, Section VIII concludes the paper.

[1] Azure (https://azure.microsoft.com/), Amazon (http://aws.amazon.com/)

II. Motivation and Challenges

Let us consider a fictitious company (UMASoft) that wants to develop a medical software solution for hospital management and administration (HospitalSoft). In order to save on deployment and maintenance costs, UMASoft decides to develop HospitalSoft as a cloud application using Microsoft Azure [13]. The intention of UMASoft is to sell HospitalSoft to different customers, so in order to have different configurations for each hospital, UMASoft will follow a multi-tenant approach, considering each hospital as a tenant. Also, HospitalSoft plans to use some of the services provided by the cloud platform Azure, such as persistence or geo-replication.

Suppose that UMASoft customers are the Spanish hospitals of Málaga, Seville and Madrid, among others. Figure 1 shows an instance of the software architecture of HospitalSoft. First, HospitalSoft application provides a set of services that are common to all the tenants from the same virtual machine (see the components inside the `Common Services Virtual Machine`). Some of them, stereotyped as «`UMASoft`», are hospital specific services, such as an appointment system (`Appointment System`) and a medical history of the patients (`Medical History`). Others, stereotyped as «`Azure`», are services offered by the Microsoft's cloud platform, such as a persistence service to store all the medical data (`SQL DataBase`) and the patients' appointments (`BlobStorage`), or the possibility to create multiple copies of the data in several data centers (`GeoReplication`).

Secondly, in addition to these common services, UMASoft provides each tenant (or hospital) a set of tenant-specific services configured to their different requirements and needs. For instance, only Málaga Hospital carries out vascular surgery, and thus the `VascularSurgery` component is included only for this tenant. Similarly, the `Nephrology` and the `Neurology` components are instantiated for Málaga Hospital and Madrid Hospital tenants but not for Seville Hospital. Finally, not only will the application specific components vary among tenants, but also some of the cloud platform services are subject to customization by the tenants. For instance, in Figure 1 we show how the authentication mechanism is different for each hospital: Málaga Hospital uses a digital certificate to authenticate the medical personnel in the system (`DigCertAuthent`) and a user-password mechanism to authenticate patients in online doctor's appointment services (`UserPassAuthent`); while Seville Hospital uses a user-password mechanism for everyone and Madrid Hospital uses digital certificates for everyone.

We can observe that normally only a subset of the application variants are deployed in each tenant. Taking into account that UMASoft has to generate and maintain hundreds of different configurations of its HospitalSoft application, implying the management of thousands of components, one important challenge is to **provide mechanisms to straightforwardly express and manage the variability of the cloud-based multi-tenant applications**. But once deployed, multi-tenant applications have to be adapted to both technological upgrades and the changing necessities of customers.

In our case study, this means that UMASoft has to manage the evolution of hundreds of HospitalSoft configurations. For instance, cloud platform services are continuously being upgraded: new authentication (e.g., social network identity authentication, biometrics) or persistence (e.g., sharding databases, affinity groups) mechanisms frequently appear. UMASoft may want to offer these new services to its customers and this implies that these services will have to be incorporated into all the tenants that require them. UMASoft can also decide to change the cloud platform provider and migrate its application to a new one (e.g., moving to Amazon Web Services). In this case, the part of the application architecture that depends on the platform's services needs to be adapted to the services offered by the new cloud platform. Finally, over the application's lifetime, UMASoft's customers may demand new functionality (e.g., a new module to manage transplant surgery or changes to the authentication mechanisms).

Considering changes that need to be performed on the software architecture of the multi-tenant application, all the aforementioned situations can be expressed with three different evolution scenarios: (1) a new component needs to be incorporated into the software architecture, either provided by the cloud platform (i.e., Azure) or implemented by the application provider (i.e., UMASoft); (2) an existing component needs to be removed from the software architecture, and (3) an existing component needs to be configured with new parameter values. But evolving a multi-tenant application implies having to calculate and perform these changes for thousands of components running on hundreds of tenants. So, the main challenges of evolving multi-tenant applications are **the automatic calculation of the changes that must be performed for each tenant** and **the automatic propagation of these changes for all the tenants at the architectural level.** Moreover, this automatic evolution process will be useful only if it is correct and efficient for a large number of tenants so we need to **demonstrate the efficiency and correctness of the evolution process.**

III. Variability Management with CVL

In this section we describe how our approach uses CVL to manage the variability of the architecture for all the tenants in a cloud-based application. As shown in Figure 2 for the HospitalSoft application, we explicitly model the commonalities and variabilities of the tenant specific functionality (i.e., the subtree under the `TenantSpecific[1..*]` feature with the functionality that can be customized for each tenant) and the common functionality (i.e., the subtree under the `CommonServices` feature with the functionality that all tenants share).

The CVL variability model includes: (1) an abstract part (the tree structure on top of Figure 2) with the *variability specifications* (VSpecs, or features) of the application's functionality and the cloud-platform's services, and (2) a realization part (middle of Figure 2) with the *variation points*. The abstract part specifies the relationships between the features,

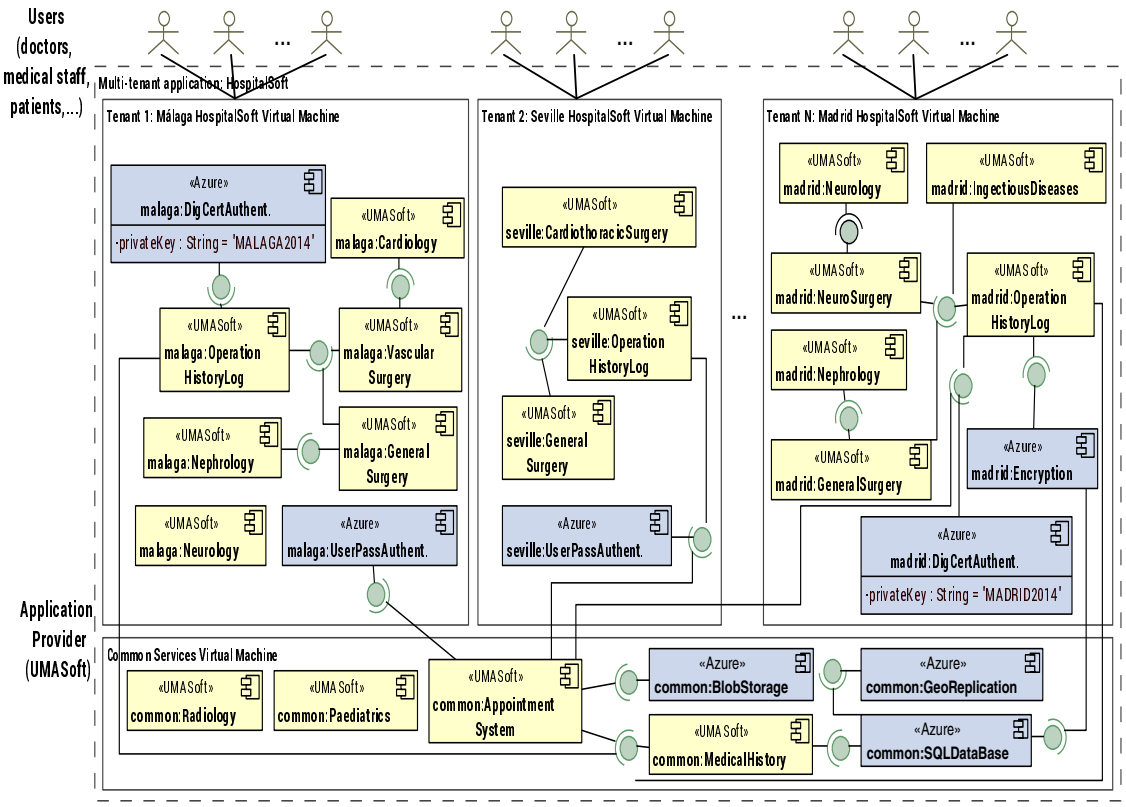

Fig. 1: Software architecture of a multi-tenant application in Microsoft Azure.

those that are optional and those that are mandatory, and the constraints between them. An application *configuration* consists of a set of features selected (or resolved) in the tree that respects the set of tree and cross-tree constraints. Variation points represent the specific modifications, as model-to-model (M2M) transformations, to be done in the architecture when a particular feature is selected in a configuration. So, each variation point is bound to a feature in the tree and has one or more references to elements of the architecture.

In order to support the specification of different configurations for each tenant, our approach defines the tenant specific functionality under a clonable feature (`TenantSpecific[1..*]` in Figure 2). The clonable feature has a cardinality [1..*] that indicates that this feature can be instantiated one or more times and then its subtree can be resolved differently for each instance. In our example, the cardinality represents the number of tenants, and each instantiation of the `TenantSpecific[1..*]` feature defines the configuration for a particular tenant (e.g., Seville hospital in the case study).

By selecting the appropriate features under the `TenantSpecific[1..*]` clonable feature, and executing CVL, our approach creates the architecture configuration shown in Figure 1, with each tenant configured to their needs.

IV. Evolution Management with CVL

Once a specific architecture configuration customized to each tenant has been generated and deployed, the multi-tenant application is susceptible to evolution due to both technological upgrades and the changing necessities of customers. Coming back to our HospitalSoft application, let us suppose Microsoft incorporates a new functionality in its cloud platform: a recovery service, an authentication mechanism based on Facebook and a persistence mechanism. UMASoft wants to provide these new services to its tenants (i.e., the hospitals).

As the first step of our evolution process, UMASoft evolves both the variability model (new features shown in grey in Figure 2) and the application base architecture (the bottom half of Figure 2 shows the evolved software architecture) in order to incorporate the changes[2]. For instance, the `Recovery`, `SocialIdentity`, `Caching`, `MongoDB` and `Sharding` features have been incorporated in the variability model in order to add the new recovery service, the new authentication mechanism, and the aforementioned new persistence mechanisms (a new non-relational database and a new partition mechanism for database). Examples of the new components are the `Recovery`, `SocialIdentityAuth` and `MongoDB` components, among others.

The second step of the evolution process is to propagate these changes consistently in the configuration of all existing tenants (Figure 1). In order to automate this task our approach proposes dividing this second step in two main parts: (i) modifying the current configuration of all tenants, producing a new evolved configuration (**Evolve Configuration** algorithm

[2]In the example only new features are added, but other evolution scenarios are possible.

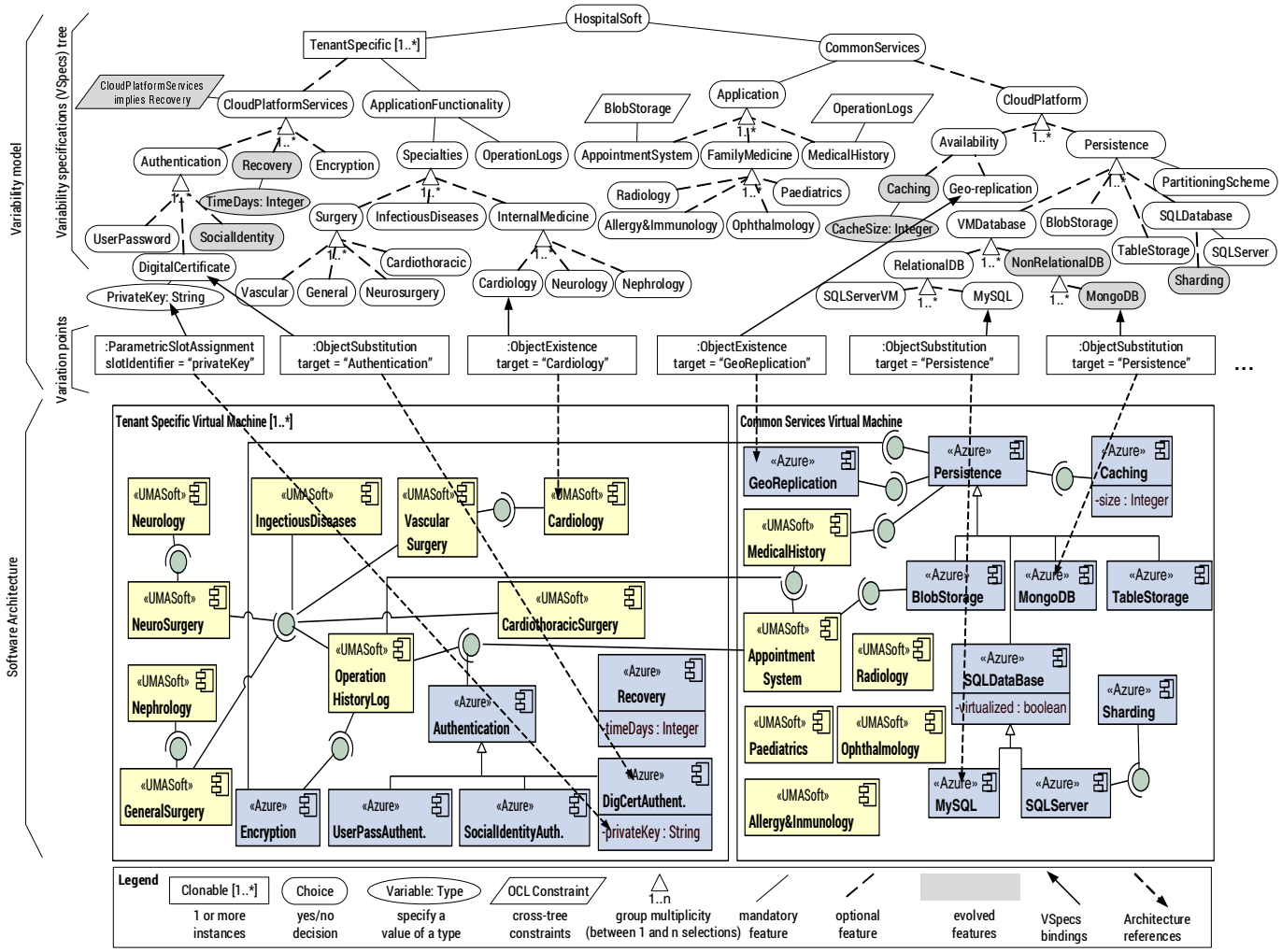

Fig. 2: HospitalSoft CVL Variability Model and Software Architecture.

of Section IV-A); and (ii) calculating the differences between the evolved configuration and the one previously deployed (**Difference Configuration** algorithm of Section IV-B).

The third and most difficult step consists in automatically propagating the evolution changes calculated by previous algorithms to the previously deployed software architecture. To do this, we define the **Create Weaving Model** algorithm (Section V) that generates a weaving model in CVL, which is in charge of applying the modifications defined previously at the feature level for the software architecture specified in MOF. The 'execution' of the weaving model will weave new components into the evolved software architecture, will unweave components from the original one, or will modify existing components by changing their parameters' values. This algorithm is one of the main contributions of the paper since, as discussed in Section 8, existing approaches that manage the evolution with SPLs (as in [14], [15]) only tackle the evolution management problem at the feature level, and require manually modifying the software architecture to reflect the calculated changes — e.g., modifying each tenant's configuration files, or defining a manual mapping between the feature model configuration and the evolved architecture for each tenant. The rest of this section describes the first two algorithms. The third algorithm is described in Section V. Finally, the CVL engine is executed with the weaving model as input in order to obtain the final architecture with the evolution changes for each tenant.

A. Evolve Configuration Algorithm

The main steps of the **Evolve Configuration** algorithm[3] are shown in Figure 3. The algorithm takes as input the previous configuration model (that will be evolved), the evolved variability model (previously upgraded by the application provider), and the new services required by each customer; and generates an evolved configuration model. The generated model represents a new valid configuration of the multi-tenant application with all the tenants' configurations evolved (i.e., the hospitals' configurations). The algorithm ensures that the generated configuration contains all the required features, taking into account the tree and cross-tree constraints defined in the variability model.

[3]The formalization of the algorithms and their complete definition can be found in http://caosd.lcc.uma.es/papers/evolutionAlgorithms.pdf.

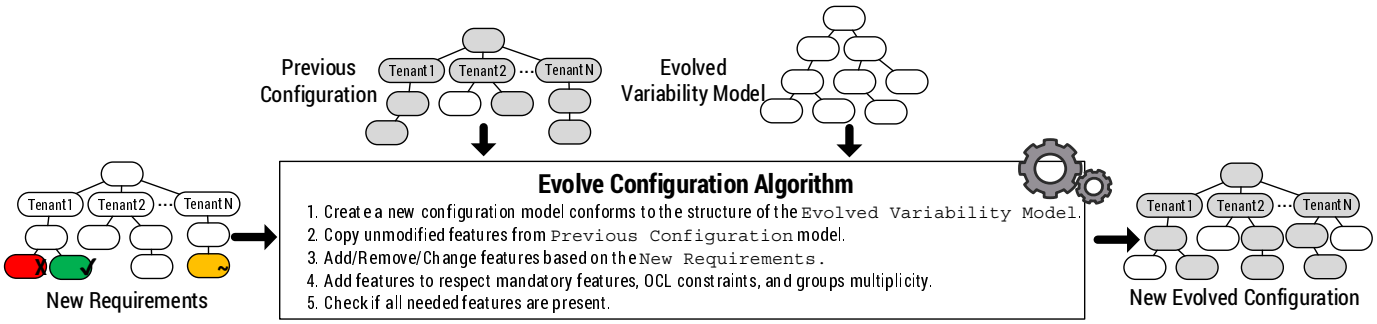

Fig. 3: Evolve Configuration algorithm.

Figure 4 shows a partial view of the three inputs of this algorithm. After initializing the output model, it copies those features that do not change from the previous configuration (see white features in Figure 4); adds the selected features (in green), does not add the unselected features (in red) and adds the modified features (in yellow); and adds those features required by the new constraints of the evolved variability model (in grey); and finally performs some validity checks of the output (i.e., the evolved configuration). For instance, for the Málaga tenant the `UserPassword` feature is removed and the `SocialIdentity` feature is added as a new requirement. Moreover, the `PrivateKey` parameter of the digital certificate is updated with a new value in both the Málaga tenant (`"MALAGA2015"`) and Madrid tenant (`"MADRID2015"`). Also, the `Recovery` feature and its `TimeDays` parameter, which specifies the time delay of the back-up, are added in all tenants due to the new constraint (`CloudPlatformServices implies Recovery`) in the evolved variability model.

Once a valid evolved configuration model is generated, the next algorithm calculates the differences between the previous configuration and the new one.

B. Difference Configuration Algorithm

The **Difference Configuration** algorithm is defined in Figure 5. The algorithm takes as input two configurations (i.e., the previous configuration and the new evolved configuration generated with the *Evolve Configuration* algorithm) and calculates the differences between them. Differences are determined by: (1) the new features selected in the new configuration that were not in the previous one (`SELECTIONS`); (2) the features of the previous configuration that are deselected in the new one (`UNSELECTIONS`); and (3) the variable features that change their values from the previous configuration to the new one (`MODIFICATIONS`).

V. Propagation of Evolution Changes to the Architecture

In this section, we define the third algorithm of our evolution approach, which generates a weaving model in CVL in order to propagate the evolution changes calculated with the two previous algorithms to the deployed architecture. Firstly, in order to precisely define the **Create Weaving Model** algorithm in CVL (Section V-C), we need to formally define the different CVL models – i.e., the variability model and configuration models. The formalization of CVL is partially done in [16]. However, the specification defined in [16] only formalizes the abstract part (i.e., the VSpecs or tree) of the variability models, but not the variation points and the configuration models. Thus, as part of our work we completed the specification defined in [16] to formalize the variation points of the variability model (Section V-A) and the configuration models (Section V-B).

A. Formalization of the CVL variation points

Variation points define the points of the architectural model that are variable and can be modified during CVL execution. They also specify how those elements are modified by defining specific modifications to be applied to the architecture by means of M2M transformations (e.g., in ATL [17]). The semantic of these transformations is specific to the kind of variation point. For instance, some of the variation points supported by CVL are the existence of elements of the architecture (`ObjectExistence`) or the links between them (`LinkExistence`), or the assignment of an attribute's value (`ParametricSlotAssignment`)[4]. An important kind of variation point is the `Opaque Variation Point` (`OVP`) that allows defining new custom model transformations that are not pre-defined in CVL. During CVL execution, the CVL engine delegates its control to a M2M engine in charge of executing the transformations defined by the variation points.

To represent the variation points of the variability model, we define a tuple: $variationPoints = (VP, type, ovptype, semantic, binding, MOFRefs)$, the content of which is:

- $\boldsymbol{VP}$. Set of finite, non-empty, unique names of variation points.
- $\boldsymbol{type : VP \rightarrow VPType}$. Function that returns the type of the variation point from the CVL variation point taxonomy.
- $\boldsymbol{ovpType : VP \rightarrow OVPType}$. Partial function that for an OVP returns its type.
- $\boldsymbol{semantic : OVPType \rightarrow SemanticSpec}$. Function that returns the semantic specification of an OVP type. This includes the transformation language and the model transformation to be executed by the M2M engine of CVL.
- $\boldsymbol{binding : VP \rightarrow VSPEC}$. Function that returns the VSpec (i.e., feature) bound to the variation point.
- $\boldsymbol{refs : VP \rightarrow P(MOFRef)}$. Function that returns the MOF references of the software architecture that is linked with the variation point.

B. Formalization of the CVL Configuration Models

Given a CVL variability model V, a configuration model R for V is a collection of VSpec resolutions ($VSPEC_{res}$)

[4]The complete taxonomy of variation points is available in [16].

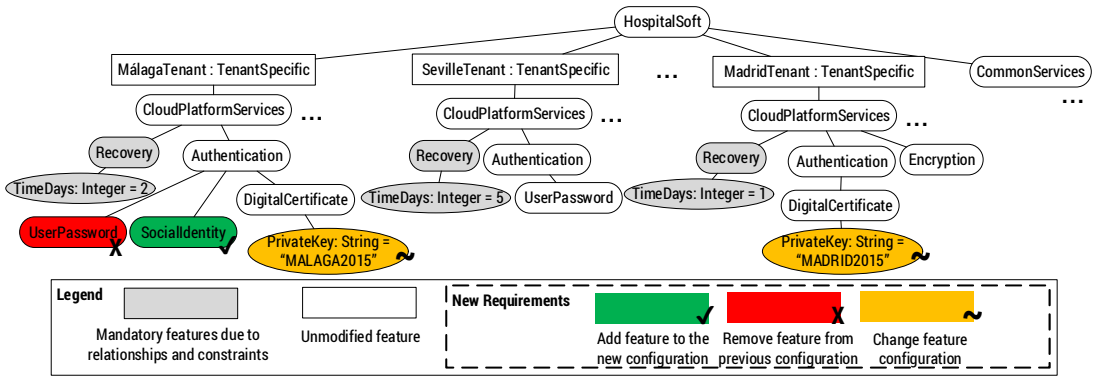

Fig. 4: Information managed by the Evolve Configuration algorithm.

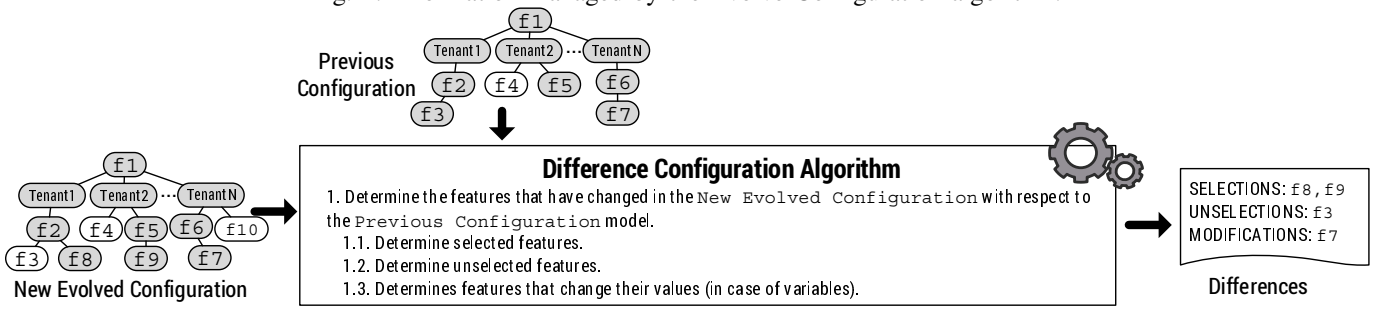

Fig. 5: Difference Configuration algorithm.

— i.e., the features selected, that can be choice resolutions ($CHOICE_{res}$), variable value assignments ($VARIABLE_{res}$), and clonable instances ($CLASSIFIER_{res}$), where each resolution in R resolves exactly one VSpec in V. Each kind of VSpec has its own kind of resolution: choices are resolved by deciding them negatively or positively; variables are resolved by providing a value; and clonables are resolved by instantiating them. The formalization of a configuration model R mirrors the formalization of V and extends it with the following definitions:

***VSPEC*$_{res}$.** Collection of finite, unique names of VSpec resolutions. $VSPEC_{res}$ is partitioned into $CHOICE_{res}$, $VARIABLE_{res}$, and $CLASSIFIER_{res}$. For example, $CLASSIFIER_{res}$ will include all instances of the `TenantSpecific` clonable feature, with a different prefix for each instance (e.g., `MálagaTenant:TenantSpecific`). The same prefix is used for the children (e.g., `MálagaTenant:Authentication`).

***resolved* : *VPSEC*$_{res}$ → *VSPEC*.** Function that for a given VSpec resolution (e.g., `MálagaTenant:Authentication`) returns the resolved VSpec of the variability model (e.g., `Authentication`).

***decision* : *CHOICE*$_{res}$ → *Boolean*.** Function that for a given choice resolution returns *true* if it was decided positively, or *false* if it was decided negatively.

***value* : *VARIABLE*$_{res}$ → *Value*.** Function that for a given variable returns the value assigned in its resolution.

C. Create Weaving Model Algorithm

The **Create Weaving Model** algorithm is defined in Figure 6. The algorithm takes as input: (1) the new evolved configuration (obtained from the *Evolve Configuration* algorithm), (2) the differences between the previous configuration and the new evolved one (obtained from the *Difference Configuration* algorithm), and (3) the base software architecture of the application and the already deployed architecture with the previous configuration of the components. The generated output is a weaving model in CVL with the information about the specific architectural elements that need to be added, removed, and/or configured in the existing architecture configurations, and the particular model transformations to be executed with the CVL runtime engine to perform the required changes. Thus, the algorithm does not directly generates the evolved software architecture, but the evolution rules that have to be 'executed' to evolve the architecture.

The algorithm extends the evolved configuration model to include the differences as new selected features in the configuration of the weaving model (lines 2 and 3). This is necessary because CVL only executes those variation points associated with features resolved positively (i.e., selected in a configuration), and unselected features in the new evolved configuration will require unweaving (i.e., remove) some elements from the architecture. Then the algorithm generates a variation point for each difference in the configuration (lines 5 to 32) associating the appropriate type of variation point with the kind of change required. Each variation point is a tuple like the one formalized in Section V-A. For instance, in order to substitute an existing element in the architecture we use the `FragmentSubstitution` variation point (lines 13 to 15); to add a new element to the architecture we use an `OVP` with a custom model transformation in charge of performing the weaving (lines 16 to 21), which may be different for each feature (line 17); to update the value of a variable we use the `ParametricSlotAssignment` variation point (lines 22 and 23); and finally to remove an existing element in the architecture we use another `OVP` with a model transformation to perform the unweaving operation (lines 25 to 29).

Finally, the weaving model obtained is executed by the CVL engine in order to generate the evolved architecture with the required changes for each tenant. An additional step that is required in order to integrate our approach with current cloud platforms is the generation of the scripts or configuration files

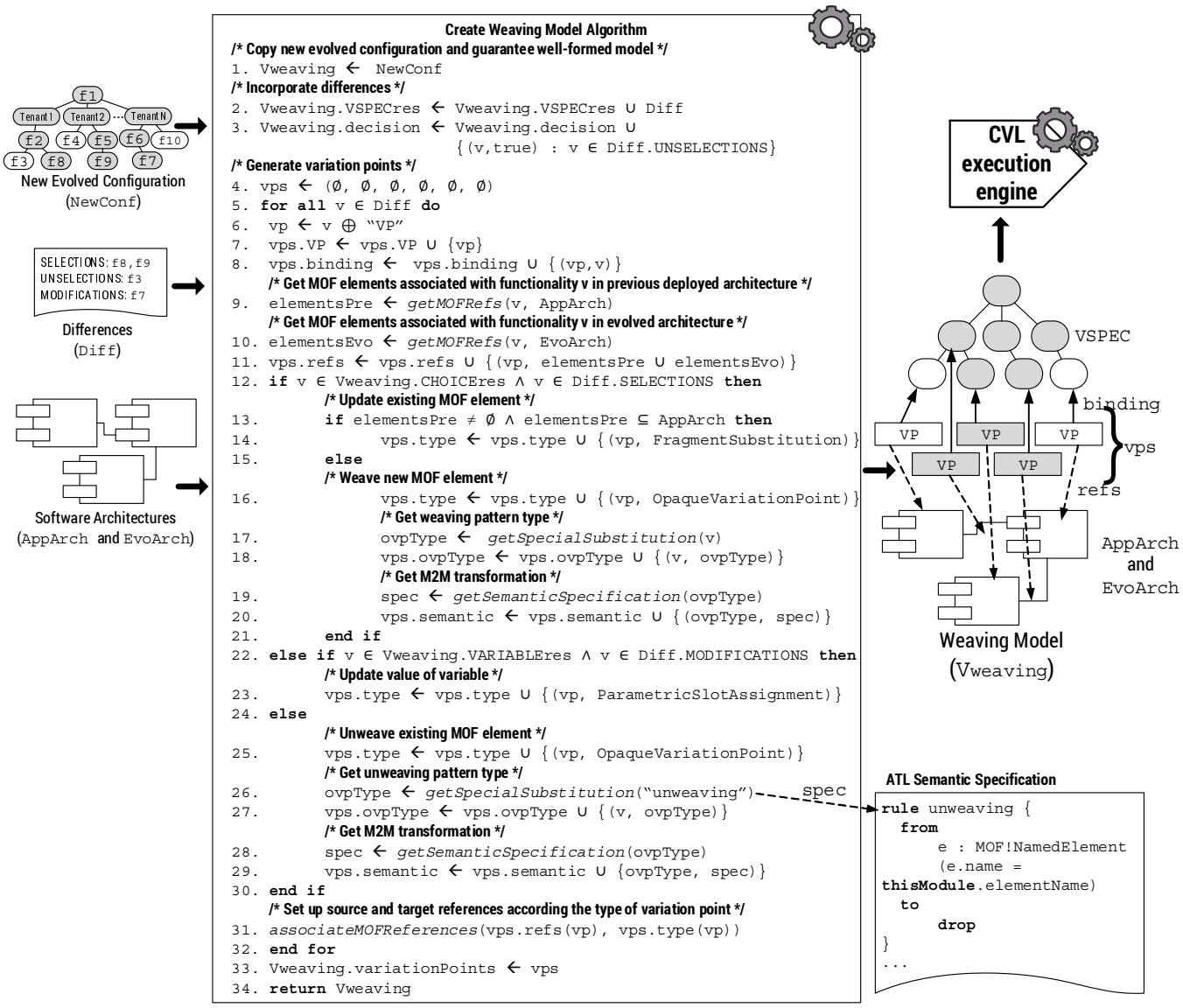

Fig. 6: Create Weaving Model algorithm.

required by each platform. They are different for each platform and are implemented by adding new OVPs with platform-specific model-to-text transformations. The output will be the configuration files required by the platforms. Notice that the reconfiguration itself is performed by the mechanisms already provided by the cloud platforms.

VI. Evaluation

We evaluate the efficiency and correctness of our evolution approach by calculating the time complexity of the algorithms presented in Sections IV and V, and by modeling the CVL models as Constraint Satisfaction Problems (CSPs). Also, a Java implementation of the algorithms is provided[5].

A. Efficiency of the evolution algorithms

The computational efficiency of an algorithm is the number of basic operations it performs depending on its input length [18]. To evaluate the efficiency of the evolution algorithms we analyze the time complexity of them. Let us consider the following basic operations: (i) the set union with a single element that corresponds to the addition of a feature to the variability or configuration model; (ii) the set difference with a single element that corresponds to removing a feature from the model; and (iii) checking whether or not an element is in a set. Formally, let A be a set of features (VSpecs) and x a single feature (i.e., a simple choice, variable, or clonable feature). The input size of our evolution algorithms is the size of the variability model (m, the number of features) and the size of the configuration model (n, the number of resolved features). The size of the configuration model depends on the number of tenants (t) — i.e., t is the number of instances resolved of the clonable features in a configuration model. To simplify, we can consider $n = m \times t$ the worst case of the configuration model size, in which all possible resolutions for a clonable feature are resolved. Normally, $n \leq m \times t$ due to

[5]Code available in http://150.214.108.91/code/cvl and http://150.214.108.91/code/cvltool.

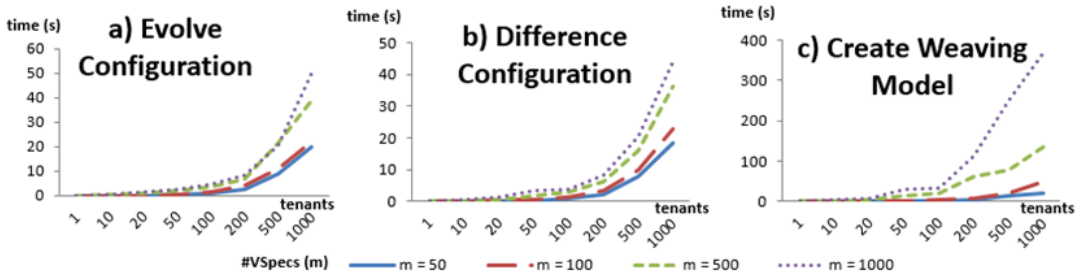

Fig. 7: Efficiency of the evolution algorithms.

TABLE 1: Complexity of the evolution algorithms.

Algorithm	Computational complexity
Evolve Configuration (Figure 3)	$\mathcal{O}(5n^2 + \frac{3}{t}n^2 + (9 + t + \frac{1}{t})n)$
Difference Configuration (Figure 5)	$\mathcal{O}(4n^2 + (8 + t)n)$
Create Weaving Model (Figure 6)	$\mathcal{O}(3n^3 + 18n^2 + 3n)$

the tree and the cross-tree constraints.

For the *Evolve Configuration* algorithm (Figure 3) the worst case corresponds to the inputs with a previous configuration of size $n = m \times t$ and the new requirements input with size also $n = m \times t$ which means changes in all the features. The efficiency of this algorithm can be easily captured following the Big $\mathcal{O}$ notation. For instance, creating a new configuration model conforming to the structure of the evolved variability model (line 1 in Figure 3) takes $\mathcal{O}(m \cdot n)$ operations. Then, copying the previous configuration (line 2) takes $\mathcal{O}((3 + \frac{1}{t})n^2 + t \cdot n)$ operations in the worst case[6]. Following a similar analysis for the rest of the algorithm, the complete *Evolve Configuration* algorithm has $\mathcal{O}(n^2)$ complexity. Table 1 shows the computational efficiency in terms of the Big $\mathcal{O}$ notation for our three evolution algorithms. The first two algorithms have quadratic time complexity ($\mathcal{O}(n^2)$), while the third algorithm has cubic time complexity ($\mathcal{O}(n^3)$) in n.

Figure 7 shows the results of the empirical experiments for the three evolution algorithms. The efficiency depends on the number of tenants (t) and on the number of features in the evolved variability model (m). The experiments were done on a laptop Intel Core i3 M350, 2.27GHz, 4 GB of memory, and with 1.7 JVM. For instance, to evolve a configuration model for a case study with 1000 tenants and 1000 features resolved for each tenant, the *Evolve Configuration* algorithm takes around 50 seconds. To calculate the difference between the previous and the new configuration, the algorithm takes around 45 seconds. Finally, creating the weaving model requires around 6 minutes in the worst case — i.e., when all features change, and thus, variation points must be defined for all features. We can conclude that the efficiency is acceptable for huge models (e.g., models with a million of features).

B. Correctness of the evolution algorithms

In order to demonstrate the correctness of the evolution algorithms (i.e., check whether the algorithms generate a valid configuration model that satisfied the new evolved variability model), we have modeled the variability model and the configuration models using Choco[7], a Java library for Constraint Satisfaction Problems (CSP) [19]. A CSP is defined by a triplet (X, D, C), where X is a set of *variables*, D is a set of *domains* for the variables, and C is a set of *constraints*. We map the CVL variability model to those concepts: (i) variables are the features of the variability model; (ii) the domain is $\{0, 1\}$ that corresponds with the semantic of the resolved or not resolved feature; and (iii) the constraints include the tree and cross-tree constraints. Choco allows the automatic generation of the set of the minimal full resolutions that satisfy a set of initial constraints. To do that, we define an objective function in CSP ($\sum_{i=1}^{n} v_i$) that minimizes the number of resolutions — i.e., the number of variables with value 1. We have checked that the output of our algorithms correspond to one of the tuples $v = \{v_1, \ldots, v_n\}$ generated by Choco.

C. Discussion

We believe that evaluating the time efficiency and correctness of the evolution algorithms is the first step to be able to apply our approach. However, there are other important measures that need also to be taken into account and are considered as threats to validity. For instance, although the approach is scalable from the point of view of the time consumed by the algorithms, also the scalability regarding the space requirements needs to be considered. Moreover, an experimental study to quantify the usefulness of our approach — i.e. the value added to the developers of multi-tenant applications, would be very interesting. Other measures, such as the maintenance and transition cost would provide important information toward the use of our approach.

VII. Related Work

Figure 8 summarizes how existing approaches about evolution of multi-tenant applications cope with the challenges identified in Section II.

Approaches that manage the variability of multi-tenant applications use either Feature Models (FMs) in an SPL [4], [14], [15] or reference architectures [9], [20] (see first column in Figure 8). FMs have the advantage of being a well-known technique supported by many tools. However, the main shortcoming of FMs is that an additional process is required to establish the relationship between the abstract variability and the elements of the architecture. In [14], an external Variability

[6]See complete algorithms in http://caosd.lcc.uma.es/papers/evolutionAlgorithms.pdf

[7]http://choco-solver.org/

	Challenge 1: Manage the variability of Cloud Multi-Tenant Applications	**Challenge 2**: Manage the evolution of Cloud Multi-Tenant Applications	**Challenge 3**: Automatically propagate the evolution changes at the architectural level
Cavalcante et al. [4]	SPL, no multi-tenant, no cloneables. Extended feature models with attributes (e.g., pricing) of the cloud platform services.	No evolution management.	No evolution changes. Conditional compiling technique (e.g., preprocessor directives) at code level to map features to functionalities.
Kumara et al. [5, 23]	Service-Oriented DSPL that supports runtime sharing and variations in a single application instance. Realization of the variability through a compositional approach [5, 23].	Not automatic. Changes done in each layer of the SPL. Identification of the potential impacts of a change on the tenants [23].	Automatically by defining a script that is manually designed [23].
Baresi et al. [6]	Dynamic SPL: CVL to augment BPEL processes with variability.	No evolution management. Reconfiguration of services.	Changes are directly done intrusively in the running process instances using AOP.
Mietzner et al. [8]	Only variants for non-functional properties (e.g., titles, workflows,...) using Orthogonal Variability Model (OVM). No cloneables.	No evolution management. Configurations need to be updated for each individual tenant.	Not automatic. Annotations in the variability model with deployment information and configuration files for each tenant.
Yang et al. [9]	No variability management. No multi-tenant. Relationships between quality attributes and architecture patterns.	Not automatic. Evolution driven by the non-functional requirements (e.g., performance, efficiency,...).	No evolution changes. Automatic selection of an architectural pattern based on quality metrics.
Nguyen et al. [10, 22]	Feature-Oriented approach to modeling and manage variability in process-based service compositions [10, 22].	No evolution management. Reference architecture and WSVL language to extend and customize services [22].	Use of the WSVL language to map features to services in the architecture [22].
Gamez et al. [14]	SPL: cardinality-based feature models with Hydra Tool. Cloneable features to configure different sensor devices.	Automatic only at feature level: differences and create configuration operators for simple features (no variables).	Not automatic. A Variability Modeling Language (VML) [21] is required to map features to actions over a reference PLA.
AbuMatar et al. [15]	SPL: variability framework with feature models and a multi-view meta-model. No multi-tenant management at features level.	Automatic single tenant evolution. Model Driven Engineering (MDE): model derivations from the meta-model for each tenant.	Automatic, but model derivation for single tenant. Meta-model's relatioships and OCL rules to map features and architecture.
Wu et al. [20]	No variability model, no multi-tenant, no cloneables. Industrial PLA: Wingsoft Examination System Product Line (WES-PL).	Not automatic. Evolution as small revisions (e.g., merges, derivations,...) to a reference skeleton architecture.	No evolution changes. Linear evolution: generation of a new product from small revisions to the reference architecture.
Our approach	**We model the variability of the software architecture of cloud multi-tenant applications using cardinality-based variability models with CVL [12].**	**We define a process to automatically generate the evolution changes over the software architecture of cloud multi-tenant applications.**	**We propagate the changes required to evolve the multi-tenant application automatically, by modifying the software architectures of all the tenants.**

Fig. 8: Our challenges and the state of the art.

Modeling Language (VML) [21] is used to map the features to actions over the software architecture of a sensor network. VML is dependent on the architectural language used to model the software architecture, and thus, a custom VML file with the mapping information needs to be manually created for each different FM and each different architectural language.

Another point to take into account is that most of the existing approaches focus on modeling the variability of non-functional properties in multi-tenant applications, such as the pricing of the services and the service availability, as in [4] and [8], or focus on analyzing how variations in the services affect the overall quality attributes of the architecture (e.g., performance, efficiency, etc.) as in [9]. But, they do not address the variability modeling of the functional components of the architecture (e.g., the variability of an encryption or authentication component) as we propose. Gamez et al. [14] is another approach that models the variability of functional components in architectures as we do, but it focuses on sensor networks and not multi-tenant applications. Although their approach could be extended to multi-tenant applications, it requires manually defining a new VML file for each multi-tenant application. Wu et al. [20] use an industrial reference PLA that does not explicitly model the variability. Moreover, this approach requires the variability and configurations for each individual tenant to be managed independently. The same occurs in [15]. *In contrast, using CVL [12], our approach avoids the necessity of using external languages to relate the FMs and the software architecture because CVL already provides the required support to establish the links between the abstract variability specifications and the elements of the software architecture. This makes it easier to apply the specified variations over the elements of the architecture automatically, and ensures that the architecture configurations fulfill the variability specification. In addition, CVL is MOF-based and this means that any MOF-compliant architectural language can be used with the variability model.*

Baresi et al. [6] also use CVL to augment Business Process Execution Language (BPEL) processes with variability in order to generate a Dynamic SPL in cloud applications. They focus on the reconfiguration of processes at runtime by using Aspect-Oriented Programming (AOP). Nguyen et al. [10], [22] define a Web Service Variability Description Language (WSVL) to consider variability, configurations, mapping to architectures, and customization of cloud applications. However, none of these work deal with the evolution of multiple tenants, focusing on the reconfiguration of services.

Dealing with the evolution management of multi-tenant applications, Gamez et al. [14] propose automatically updating the previous configurations for all the tenants (sensor devices in their approach) when the application requirements evolve, although they only automatize the evolution process at the FM level and for simple features (i.e., yes/no decision features that do not support the evolution of variable domain features). Kumara et al. [5], [23] address the runtime evolution of single-instance multi-tenant applications and focus on controlling the impacts of a change. Changes are realized on the runtime model of the SPL based on models@runtime technology. Other approaches that also manage the evolution of cloud applications require evolving each tenant configuration individually, for example by applying small changes (i.e., revisions) to a reference architecture [20] or through model derivations from a meta-model with the variability information [15]. *As*

*part of our PLA, the first two evolution algorithms (**Evolve Configuration** and **Difference Configuration**) allow us to reason about the architectural elements (e.g., components, interfaces, relationships, and parameters) that need to be added, removed, or reconfigured for each tenant, and analyzing the evolution effort when the application evolves.*

The evolution management of a multi-tenant architecture requires propagating the changes from the variability model and configuration model to the application architecture deployed for each tenant. Manually updating these architecture configurations is a hard and error-prone task. Some approaches use configuration files or scripts that usually need to be manually updated for each tenant, as in [8], [23]. Other approaches [15] generate a new architecture from scratch, which can be highly inefficient for hundreds of tenants. *In our approach, in order to propagate the evolution changes to the architecture, a weaving model in CVL is automatically generated. This weaving model contains information about the specific architectural elements that need to be added, removed, and/or reconfigured in the existing architecture configurations, and the particular model transformations to be executed with the CVL runtime engine to perform the required changes.*

Taking into account industry solutions, the OSGi (Open Service Gateway Initiative) for Cloud[8] is a Java framework for developing and deploying modular software with support for dependency management and resolution. The framework focuses on dynamic deployment and automatic management of modules in multi-tenant applications. In addition, Salesforce[9] is a market leader in customization of enterprise multi-tenant solutions. Salesforce provides a metadata-driven architecture platform in which all customizations of the tenants (e.g., code, configuration, applications, etc.) are specified as metadata. The metadata is kept in a layer separate from the services layer, which allows seamless, easy upgrades. So, the evolution management is driven in the metadata layer as updates of this metadata. *Our approach does not try to displace or compete with these solutions, but our approach can be seen as an additional layer that serves to the software architects in the variability management and its evolution, when applications needs to maintain multiple configuration for the tenants. This is independent from the cloud platform where the applications are deployed.*

VIII. Conclusions and Future Work

We have presented an approach that uses CVL and cardinality-based variability models to manage the variability and evolution of a large number of tenants in the context of cloud applications. Evolving thousands of configurations in multi-tenant applications is a unaffordable task to tackle manually. So, we have presented three algorithms to automatically evolve a previous configuration of the tenant, calculate the changes that need to be made to the application architecture, and propagate the evolution changes to the multi-tenant architecture by applying M2M transformations. We have formalized the CVL models as a CSP to demonstrate the correctness of the algorithms, and we have also analyzed the efficiency of the algorithms.

As part of our ongoing work we plan to extend our approach to automatically deploy the evolved architecture in the specific cloud platforms, or to automatically generate the appropriate configuration files [8], scripts [23], or metadata (in case of Salesforce) from the evolved architecture. Additionally, there are other important issues of our approach that need to be evaluated as part of our on-going work, such as the usefulness, maintenance efforts, scalability and space requirements, among others.

[8]https://www.osgi.org/developer/design/cloud/

[9]https://developer.salesforce.com/page/Multi_Tenant_Architecture

Acknowledgment

This work is supported by the project Magic P12-TIC1814 and by the project HADAS TIN2015-64841-R (co-financed by FEDER funds).

References

[1] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "A view of cloud computing," *Commun. ACM*, vol. 53, no. 4, pp. 50–58, 2010.

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

[3] K. Pohl, G. Böckle, and F. J. v. d. Linden, *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag New York, Inc., 2005.

[4] E. Cavalcante, A. Almeida, T. Batista, N. Cacho, F. Lopes, F. C. Delicato, T. Sena, and P. F. Pires, "Exploiting software product lines to develop cloud computing applications," in *Software Product Line Conference*, ser. SPLC, 2012, pp. 179–187.

[5] I. Kumara, J. Han, A. Colman, T. Nguyen, and M. Kapuruge, "Sharing with a difference: Realizing service-based saas applications with runtime sharing and variation in dynamic software product lines," in *Conference on Services Computing (SCC)*, 2013, pp. 567–574.

[6] L. Baresi, S. Guinea, and L. Pasquale, "Service-oriented dynamic software product lines," *Computer*, vol. 45, no. 10, pp. 42–48, 2012.

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

[8] R. Mietzner, A. Metzger, F. Leymann, and K. Pohl, "Variability modeling to support customization and deployment of multi-tenant-aware software as a service applications," in *Principles of Engineering Service Oriented Systems*, 2009, pp. 18–25.

[9] 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.

[10] T. Nguyen, A. Colman, and J. Han, "Modeling and managing variability in process-based service compositions," in *International Conference on Service-Oriented Computing*, ser. ICSOC, 2011, pp. 404–420.

[11] J. Bosch, *Design and use of software architectures: adopting and evolving a product-line approach*. Pearson Education, 2000.

[12] O. Haugen, B. Moller-Pedersen, J. Oldevik, G. Olsen, and A. Svendsen, "Adding standardized variability to domain specific languages," in *SPLC*, 2008.

[13] D. Betts, A. Homer, A. Jezierski, M. Narumoto, and H. Zhang, *Moving Applications to the Cloud on Windows Azure*, 3rd ed. Microsoft patterns & practices, 2013.

[14] 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.

[15] M. Abu Matar, R. Mizouni, and S. Alzahmi, "Towards software product lines based cloud architectures," in *IEEE IC2E*, 2014, pp. 117–126.

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

[17] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev, "ATL: A model transformation tool," *Sci. Comput. Program.*, vol. 72, no. 1–2, pp. 31–39, 2008.

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

[19] E. Tsang, *Foundations of constraint satisfaction*. Academic press London, 1993, vol. 289.

[20] Y. Wu, X. Peng, and W. Zhao, "Architecture evolution in software product line: An industrial case study," in *Top Productivity through Software Reuse*, 2011.

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

[22] T. Nguyen, A. Colman, and J. Han, "Enabling the delivery of customizable web services," in *International Conference on Web Services (ICWS)*, 2012, pp. 138–145.

[23] I. Kumara, J. Han, A. Colman, and M. Kapuruge, "Runtime evolution of service-based multi-tenant SaaS applications," in *Service-Oriented Computing*. Springer, 2013, pp. 192–206.