Towards Software Product Lines Based Cloud Architectures

Mohammad Abu Matar[(1)], Rabeb Mizouni[(2)], and Salwa Alzahmi[(2)]
[(1)]Eitsalat British Telecom Innovation Center Abu Dhabi, United Arab Emirates
[(2)]Khalifa University of Science, Technology and Research, Abu Dhabi, United Arab Emirates
{mohammad.abu-matar, rabeb.mizouni, salwa.alzahmi}@kustar.ac.ae

ABSTRACT—Cloud computing has emerged as a model for utility computing that promotes on-demand scalability, flexible application deployment and reuse. Software product lines (SPL) promote reusable application development for product families. As any computing system, cloud-based systems evolve to respond to changing clients' requirements. Cloud-based applications can be modeled as Software-as-a-Service (SaaS) families similar to the SPL products. As SPL development techniques rely on feature models to describe the commonality and variability of family member applications, such techniques can be used to model variability in SaaS. In this paper, we describe a unified and systematic framework for modeling cloud services in a vendor-neutral manner. In addition, we demonstrate the applicability of the variability framework for building and customizing SaaS multitenant applications. Our approach is based on a meta-model that formalizes the multiple views of service-oriented SaaS applications. A proof of concept tool that automatically generates multitenant applications (to adapt to changing requirements of tenants) is presented. Our approach facilitates development of cloud SaaS families in a systematic, consistent, and platform independent way. ***Keywords—Cloud Computing; Software Product Lines; Service Oriented Architecture; SaaS***

I. INTRODUCTION

Cloud computing has emerged as a model for utility computing that promotes on-demand scalability, flexible application deployment and reuse [1]. One of the major benefits claimed for cloud computing is the flexible deployment of IT solutions that can react to changing business requirements quickly and economically. The cloud computing ecosystem consists of service providers, service consumers, and service brokers [2].

Cloud services are consumed by many clients that have different requirements. More specifically, Software as a Service (SaaS) applications are provided to diverse clients, i.e. tenants, that usually share the same code base and software instances, e.g. SalesForce CRM SaaS [15]. It should be noted that multitenant applications (MTAs) are different from multiuser applications in that multitenant systems use the same instance of the provided software, whereas multiuser systems use multiple instances of the software [16]. In turn, SaaS tenants allow their users to tailor the applications to their needs. Thus, MTAs exhibit multiple levels of variability. Varying requirements usually necessitate varying software architectures. Therefore, both requirements and architectures have intrinsic variability characteristics.

Software architecture can describe application designs from different perspectives [3]. In other words, the same application architecture consists of multiple depictions of different perspectives, also called views, which address specific architectural concerns. Managing variability in SaaS MTAs is complicated by the presence of interrelated views in these systems. A change in one view necessitates a change in other views. Therefore, manual development of variable SaaS MTAs is unwieldy and error-prone.

To achieve economies of scale, MTAs are usually deployed on top of virtualized hardware infrastructures (IaaS) and platform solution stacks (PaaS). This in turn adds another level of variability due to the varying needs of tenants at these layers too.

We categorize the different levels of variability in cloud systems as follows:

- Application Variability – SaaS tenants have varying functional and behavioral requirements in addition to the core SaaS application. Furthermore, tenants' users may require further customization in the structure and look-and-feel of user interfaces [17].
- Business Process Variability where the business workflow may vary based on specific tenants' business requirements.
- Platform Variability – Operating systems, Programming languages, Frameworks, and solution stacks, i.e. Platform as a Service (PaaS).
- Provisioning Variability – Hardware, Networking, Virtual Machines, and Elasticity, i.e. Infrastructure as a Service (IaaS).
- Deployment Variability – Public, Private, Community, and Hybrid clouds [2].
- Provider Variability, i.e. Amazon Web Services (AWS), Google Application Engine (GAE), and SalesForce to name a few.

To manage variability of cloud services, many [24], [25] advocate the use of Software Product Line (SPL) techniques due to the complementary benefits of both concepts. Software Product Lines (SPL) are families of software systems that share common functionality, where each member has variable functionality [4]. The main goal of SPL is the agile and rapid development of family members

by using reusable assets from all phases of the development lifecycle.

In an attempt to model variability of SaaS application families, we describe in this paper the use of SPL concepts to model the variability concerns of cloud systems by extending the multiple-view variability modeling of SOA systems proposed in [5]. The approach integrates feature modeling [4] [6] with service views using UML and SoaML [7]. Such an approach facilitates variability modeling of cloud service family architectures in a systematic and platform independent way.

At the heart of the approach is a meta-model [8] that describes variability in requirements and architectural meta-views of service oriented systems. This meta-model is extended to model architectural concerns of cloud services construction.

The main contribution of our ongoing research is the definition of a unified and systematic framework for modeling cloud services in a vendor-neutral manner that is amenable to be used in development tools. In addition, we demonstrate the applicability of the variability framework for building and customizing SaaS MTAs. As a proof of concept, a tool, built based on the proposed multi-view variability model, that automatically generates multitenant applications is presented. In addition to handle the changing requirements of tenants to develop/update the instance at design time, the runtime instance adaptation is realized using OSGI [21] bundles deployment.

The rest of the paper is structured as follows. Section 2 motivates the research. Section 3 describes variability modeling in SPLs. Section 4 describes the multiple-view cloud variability modeling approach. Section 5 describes the multiple-view cloud variability meta-model. Section 6 outlines a conceptual model for multitenant SaaS evolution. Section 7 describes a proof-concept tool. Section 8 reports on related work, and finally section 9 concludes the paper.

II. MOTIVATION

In service computing, service providers are usually decoupled from service consumers, thus consumers and providers vary independently of each other. This variation manifests itself in several ways, i.e. in changing requirements, changing architectures, and changing execution environments. The advent of cloud computing necessitated even more variability in the form of service types, deployment types, and different cloud participant roles. Thus, variability modeling is necessary to manage the inherent complexity of service-oriented cloud systems.

Unlike traditional SPLs, the variability of SaaS MTAs is not entirely known at design time due to potential tenants' customization requests. Thus, feature models have to be evolved, and hence the architecture, to realize the new requirements. It should be noted that this change has to be done while the tenants of the SaaS are up and running. In other words, instances need to adapt at runtime to requirement changes of the tenants.

It is hard to model complex and reasonably sized software systems from one perspective, e.g., structural. To manage this complexity, the software engineering community has used multiple-view modeling to model software systems from different perspectives [10].

For example, a service-oriented cloud E-Commerce SaaS could have a view that describes service contracts, service providers, and consumers. Another view could be a business process view that describes tenants' workflow of order fulfillment. Yet another view could be a service interface view that describes an Ordering service's operations and parameters. Another view could be a platform view that specifies solution stacks choices.

If a task changes in the business process view, say a task is added to allow electronic check payments; a new service interface for electronic check payment has to be added to the service interface view. Likewise, if a credit check contract is added in the service contract view, a credit check service provider gets added to provide this capability. For reasonably sized applications, changes in the interdependent views of cloud systems can quickly become unwieldy and difficult to manage.

Managing variability becomes even more challenging when it comes to multitenant applications. Each tenant has its own configuration of the application and although tenants use the same instance, the workflow of the application may be different for two tenants. Any addition of tenant or business requirements or change of existing tenants brings the need to adapt and evolve the running instance. This evolution should not interrupt ongoing services and requires zero-down time of the SaaS.

However, current cloud-based and SaaS development practices lack a systematic framework for managing variability and are typically dependent on proprietary platforms.

III. MODELING VARIABILITIES IN SPLS

In model-based software design and development, models are built and analyzed prior to the implementation of the system, and are used to direct the subsequent implementation. A better understanding of a system or SPL can be obtained by considering the multiple views [9, 10] such as requirements models, static models, and dynamic models of the system or SPL. A key view in the multiple views of a SPL is the feature modeling view [4, 6], which explicitly models SPL commonality and variability in terms of features, which are requirements or characteristics that differentiate among product line members.

A. SPL Modeling Process

A SPL development process has two main processes: a) Domain Engineering. A SPL multiple-view model, which addresses the different views of a SPL, is developed. The SPL multiple-view model, SPL architecture, and reusable components are developed and stored in the SPL reuse library. b) Application Engineering. The application developers select the required features for the individual SPL member. Given these features, the SPL model and

architecture are adapted and tailored to derive the application architecture.

B. Model Driven Engineering for SPLs

Our approach exploits the model driven engineering (MDE) [22] principles to achieve the following goals:

- Development of model-driven techniques to design cloud service-oriented SPLs.
- Automation of service-oriented product line engineering – this includes SPL Domain and Application Engineering.
- Development of a model-driven service-oriented prototype that realizes our approach.

We explain how traditional MDE concepts are adapted to cater for cloud service-oriented SPL engineering. In MDE, developers create requirements and design models first without worrying about coding and platform concerns. These models are referred to as Platform-Independent Models (PIM) [22]. Once the PIMs are verified to match business requirements, they are transformed to Platform-Specific Models (PSM) [22], which are geared towards technology platforms. Finally, code is generated from the PSMs for a targeted technology platform such as Java, or .NET.

In typical MDE approaches [22], [23] only one PIM is constructed for the entire application. Since SPLs are family of applications, this paper proposes the construction of three types of PIMs (Fig. 3):

- A Kernel PIM – this PIM models the common architectural artifacts for the whole SPL. The SPL PIM is a multiple-view PIM since it is based on the multiple-view service variability model [8].
- Tenant PIMs – these PIMs model the derived tenant applications of the SaaS SPL. That is, these PIMs contain the Kernel PIM in addition to the variations specific for each tenant.
- Multitenant PIM – this PIM is the result of the merged Tenant PIMs, which ensures that all tenants run on the same instance.

Consequently, all PIMs are either transformed to their corresponding PSMs, or directly to code using existing MDE approaches [22, 23].

We elaborate on the use of the aforementioned PIMs in the Multitenant Evolution section below.

IV. MULTIPLE VIEW CLOUD VARIABILITY MODEL

Inspired by the seminal work on '4+1' Architecture [10], we present a multiple view cloud variability model that describes the relevant views for cloud computing. The different service-oriented cloud views are grouped into multiple Requirements and Architectural views, which are formalized by multiple-view service variability meta-model. In addition, the feature model acts as a unifying view that provides the added dimension of modeling variability.

The multiple-view model consists of two types of views: service-oriented views [5], and cloud views. Service views consist of two Requirements views (Fig. 2), the *Service Contract Variability* View and the *Business Process Variability* View, and two Architectural views (Fig. 2), the *Service Interface Variability* View and the *Service Coordination Variability* View.

A. FEATURE VIEW

With the service views, it is possible to define the variability in each view and how it relates to other views. However, it is difficult to get a complete picture of the variability in the whole service architecture, because it is dispersed among the multiple views. The Feature View is a unifying view that focuses on service family variability and relates this variability to other service views. The Feature View is based on feature modeling [4] and is used to manage commonalities and differences among service family members in a SPL. Features are analyzed and categorized as common, optional, or alternative. Related features can be grouped into feature groups, which constrain how features are used by a SPL member.

We adopt the feature classification approach discussed in [18] where the feature model is composed of multiple feature layers. We define the following layers:

- Capability Features – this layer models the functional variability in the domain, which is the same as in our previous feature modeling approach [4].
- Platform Features – this layer models the solution stack choices. For example, an E-Commerce feature model (Fig. 1) could provide a choice of a LAMP (Linux, Apache, MySQL, PHP), or a WISA (Windows Server, SQL Server, IIS, ASP) solution stack features. The platform features are grouped into a <<at least-one-of feature group>> where each feature is annotated with a <<platform>> stereotype. This way the same SaaS could be implemented on one or both solution stacks. In addition, should the SaaS provider rely on a cloud Platform as a Service (PaaS) offering, the platform layer describes the available PaaS choices. For example, an E-Commerce SaaS could be implemented on Amazon's SimpleDB or RDS database services, and on Google AppEngine, or Amazon's BeanStalk web platforms.
- Provisioning Features – this layer models the infrastructural choices of CPU, Memory, Storage, and Network. For example, an E-commerce SaaS provider may choose to build its applications on a bare-metal dedicated infrastructure, or a virtualized infrastructure, i.e. IaaS, which is usually shared among other tenants. The provisioning features have attributes that specify infrastructure specification like CPU speed, RAM size, and storage size and type. In addition, features have attributes that specify elasticity thresholds. Features are annotated with the <<infrastructure>> stereotype.

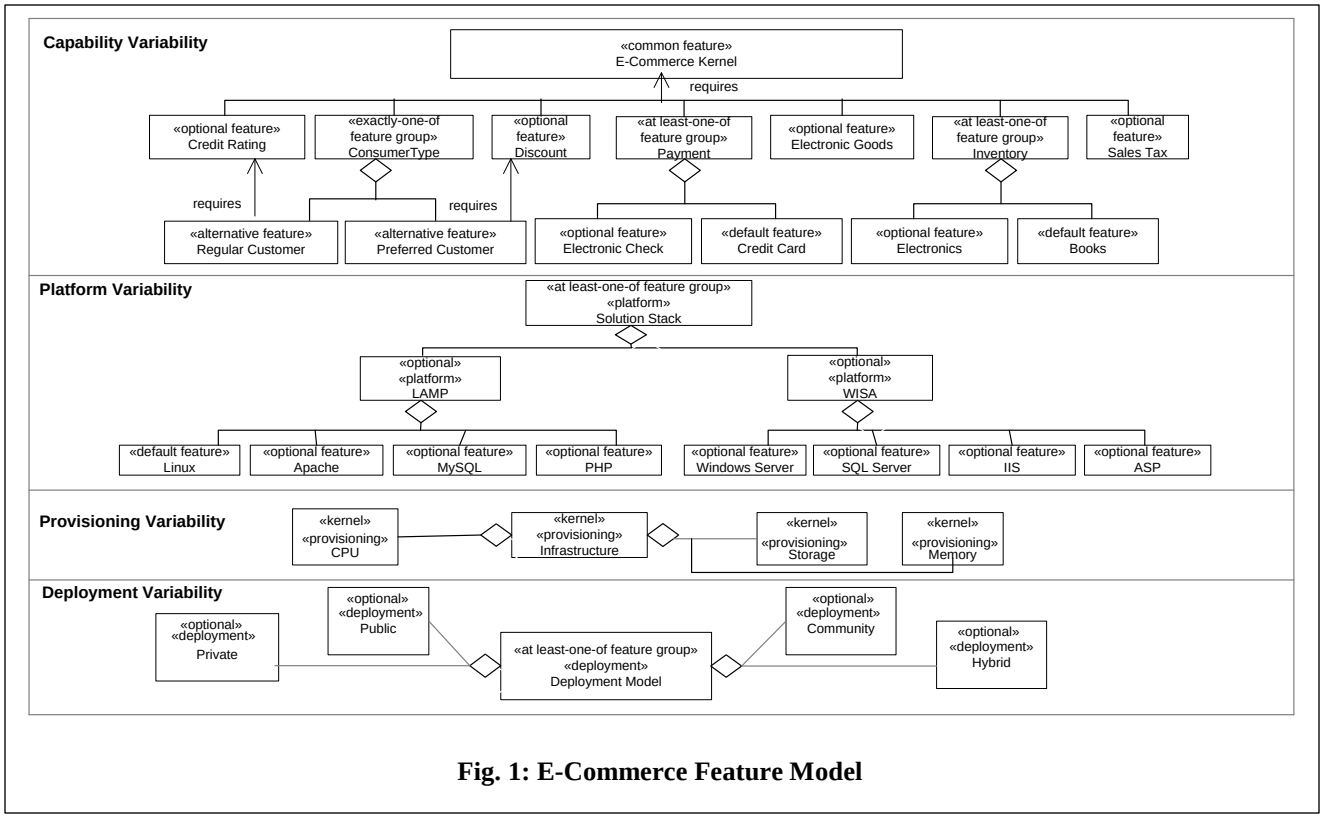

Fig. 1: E-Commerce Feature Model

- Deployment Features – Applications may be deployed on Public, Private, Hybrid, or Community clouds [1]. The features in this layer are annotated with the <<deployment>> stereotype.

B. SERVICE CONTRACT VIEW

We identify the following new Participant types: Tenant, and Cloud Provider (Fig. 2b). Tenant models SaaS enterprises each with its set of users. For example, SalesForce CRM SaaS [15] has several hundred thousand tenants. Cloud Provider models SaaS, PaaS, and IaaS providers. The Tenant element in this view has a Tenant Configuration element which resides in the Tenant Configuration view explained below. In addition, it has an NFR, i.e. non-functional requirements, element that is related to the Provisioning and Platform view explained below.

C. PROVISIONING VIEW

This view describes the hardware and/or the virtualized infrastructure associated with a certain SPL multi-tenant member application. This view is modeled by UML Deployment diagrams extended with <<infra>> stereotypes. This view consists of an Infrastructure element which consists of Server, CPU, Memory, Storage, and Network elements (Fig. 2).

D. PLATFORM VIEW

This view describes the solution stacks of the specific SPL MTA. We use UML Artifact elements, which are part of UML Deployment diagrams, extended with <<platform>> stereotypes. The main element in this view is the Solution Stack element, which consists of OS, App Server, Database, and Web Language elements (Fig. 2). Elements are categorized as kernel, or optional. Once a platform is selected, this platform becomes kernel for the SaaS instance going forward. If some tenants desire the SaaS functionality to be built on a different platform, a new instance has to be deployed. If this is an acceptable decision by the SaaS provider, it means two distinct SaaS instances will be maintained at the same time.

E. TENANT CONFIGURATION VIEW

This view describes tenants' specific feature configurations. The purpose of this view is to have a record of the constituent features in each tenant's customized version of the service. Features are added or removed from this view as the service is customized. Tenant Configuration meta-class has a Composite Structure meta-class, which contains tenant's specific parts.

V. MULTIPLE VIEW CLOUD VARIABILITY META-MODEL

The multiple-view cloud variability modeling approach is based on a meta-model [8] that precisely describes service views and views relationships. Each view in the multiple-view model is described by a corresponding meta-view in the meta-model. There are two Requirements meta-views, Service Contract and Business Process, and two Architecture meta-views, Service Interface and Service Coordination. In addition, we have the Feature meta-view that focuses entirely on variability and defines dependencies in this variability. Features are modeled by UML meta-classes that extend UML Class meta-class

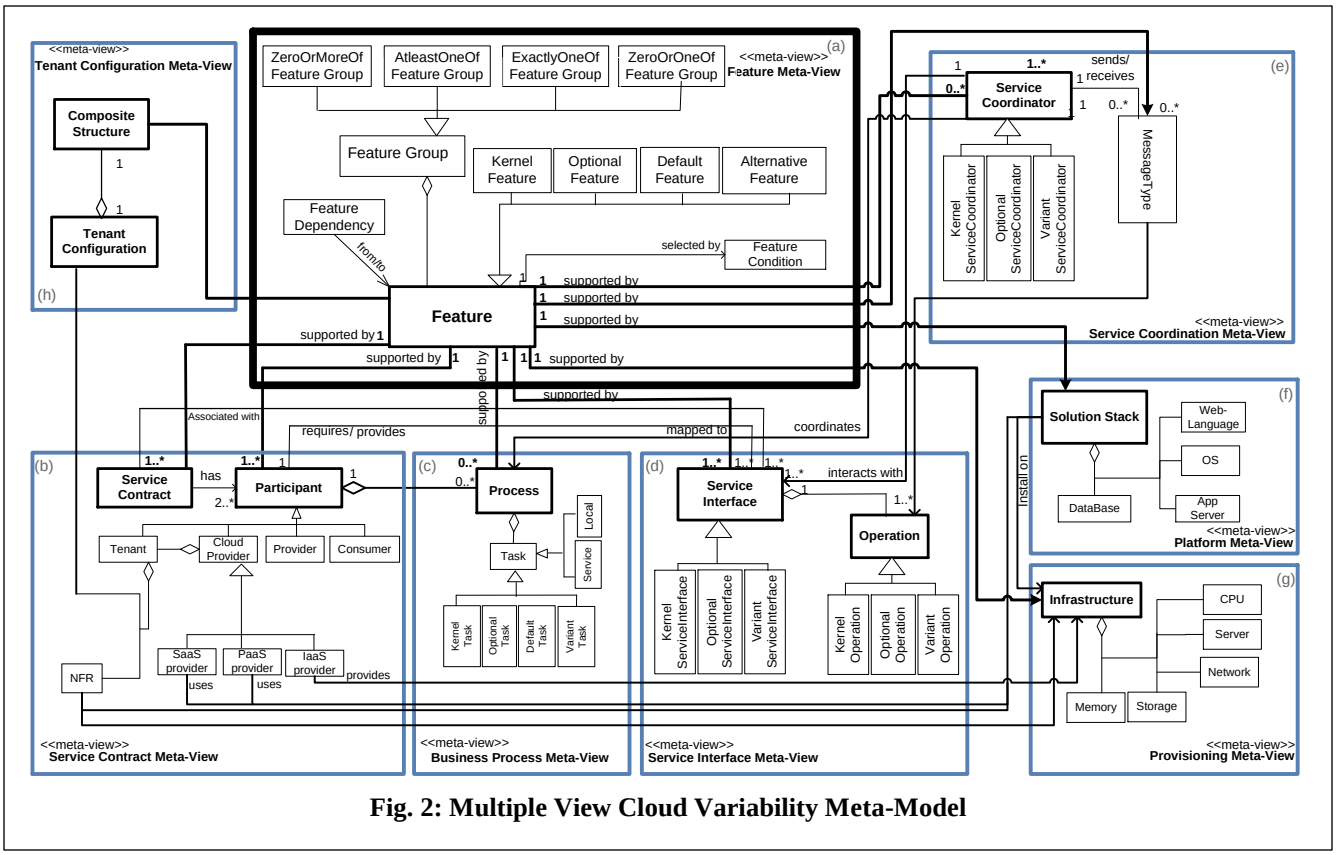

Fig. 2: Multiple View Cloud Variability Meta-Model

The meta-model (Fig. 2) consists of an extended Service Contract meta-view where the Participant meta-class was specialized with the following meta-classes: Service Consumer, Service Provider, Cloud Provider, and Cloud Tenant. Cloud Tenant meta-class consists of User meta-classes. Cloud Provider meta-class is specialized into SaaS Provider, PaaS Provider, and IaaS Provider meta-classes.

In addition, the meta-model has a new Provisioning meta-view (Fig. 2g) that consists of an Infrastructure meta-class which has Server, CPU, Memory, Storage, and Network meta-classes. This meta-view would be relevant if the SaaS Provider offers IaaS as well.

The meta-model also has a new Platform meta-view (Fig. 2f) which consists of a Solution Stack meta-class that has OS, App Server, Database, and Web Language meta-classes. This meta-view models the PaaS cloud offering and could be offered along a SaaS offering as in the case of Slaesforce SaaS and Force PaaS [15].

The meta-model defines relationships inside and between the meta-views. As in our previous meta-model discussed in section 4, all classes that are based on the meta-classes defined in this meta-model will be annotated with stereotypes respective to their meta-class names.

The meta-model also describes the inter-view relationships among the elements of the service cloud model. It is important to develop these relationships, because it enables us to analyze what happens in one view if a change happens in another view.

We assume that all SaaS MTAs are governed by Service Contracts. Service Contracts consist of at least two Participants and one or more Service Interfaces. For example, a SalesForce SaaS Service Contract could have several tenants (Participants) where each tenant has its own users (Fig. 2). In addition, a Cloud Broker could participate in the same Service Contract if SalesForce chooses to broker consumer requests. It should be noted that since a SaaS Tenant is a kind of a Participant, it could define its own business process in the Business Process view (Fig. 2). For example, a tenant could use a SalesForce CRM SaaS as part of its own business process to send existing customers some promotional materials. In addition, all Participants should provide at least one Service Interface.

An IaaS Provider, in the Service Contract View, provides infrastructure services modeled by an Infrastructure element in the Provisioning View. A SaaS provider may deploy its software on internal data centers, or lease infrastructure from a IaaS provider. This entails a relationship between the Service Contract View and the Provisioning View. Consequently, a SaaS Provider participates in a Service Contract with an IaaS provider modeled in the Service Contract View.

A PaaS Provider, in the Service Contract View, provides platform services modeled by a Solution Stack element in the Platform View. A SaaS Provider may implement its software on its own solution stacks or choose to lease a PaaS, which implies the lease of IaaS as well (Fig. 2). Again, this also implies a participation in a Service Contract with a PaaS Provider.

Feature to Architecture Mapping is governed by the meta-model's relationships [8]. In addition, the meta-model also describes relationships between the aforementioned service meta-views. Object Constraint Language (OCL) consistency checking rules are provided [8] for the relationships that cannot be explicitly described in the meta-model. The relationships in the meta-model consist of intra-view, inter-view, and feature to service variability views relationships.

Finally, SaaS member derivation, i.e. SPL Application Engineering, is also governed by the meta-model feature to architecture mapping similar to [5].

VI. MULTI-TENANT SAAS SPL EVOLUTION

In this section, we consider SaaS evolution as an example of applying our cloud variability meta-model.

As described in section 3, SPL member applications are derived based on feature selection from the feature model. However, in SaaS MTA derivation, we have to distinguish between single-instance SaaS, and multi-instance SaaS [16]. In the case of multi-instance, SaaS member derivation is similar to traditional SPL application engineering where each new tenant uses different instance of the service [4].

In the case of single configurable instance SaaS, which is the recommended model to achieve economies of scale [16], SPL application engineering becomes an SPL evolution problem, where the SaaS instance has to be customized at runtime to satisfy the requirements of new tenants and the changing requirements of the existing tenants. Additionally, this evolution problem is different from dynamic SPLs where dynamically derived member applications would be instantiated separately [17].

To demonstrate the applicability of the variability framework for building and customizing multitenant applications, we apply it to single instance multi-tenant SaaS SPL. We model tenants' evolution and we provide an initial conceptual model for tenants' dynamic construction and customization (Fig. 3). The conceptual model consists of three phases: SaaS kernel development, new tenants onboarding, and existing tenant's customization. In all phases, the starting point is the feature model (Fig. 1), which includes common and variable functionality for all tenants.

A. SaaS Kernel Development

In SaaS kernel phase, common functionality of the SaaS SPL is designed, developed, and deployed. This is similar to the 'Kernel First' approach in the SPL PLUS method [4]. Unlike [4], this kernel SaaS is a fully functional instance that offers out-of-the box functionality to customers. The kernel SaaS is architected as a kernel Platform Independent Model (PIM) [19]. Subsequently, the kernel PIM is transformed to a Platform Specific Models (PSM) [19] based on the desired target technology.

It should be noted that the kernel SaaS will be part of all future SaaS evolution phases. In other words, the SaaS kernel instance will be adapted to accommodate tenants' onboarding and customization as described below. This way, the kernel instance serves as the seed for the single instance MTAs. Further, all evolution steps will always be

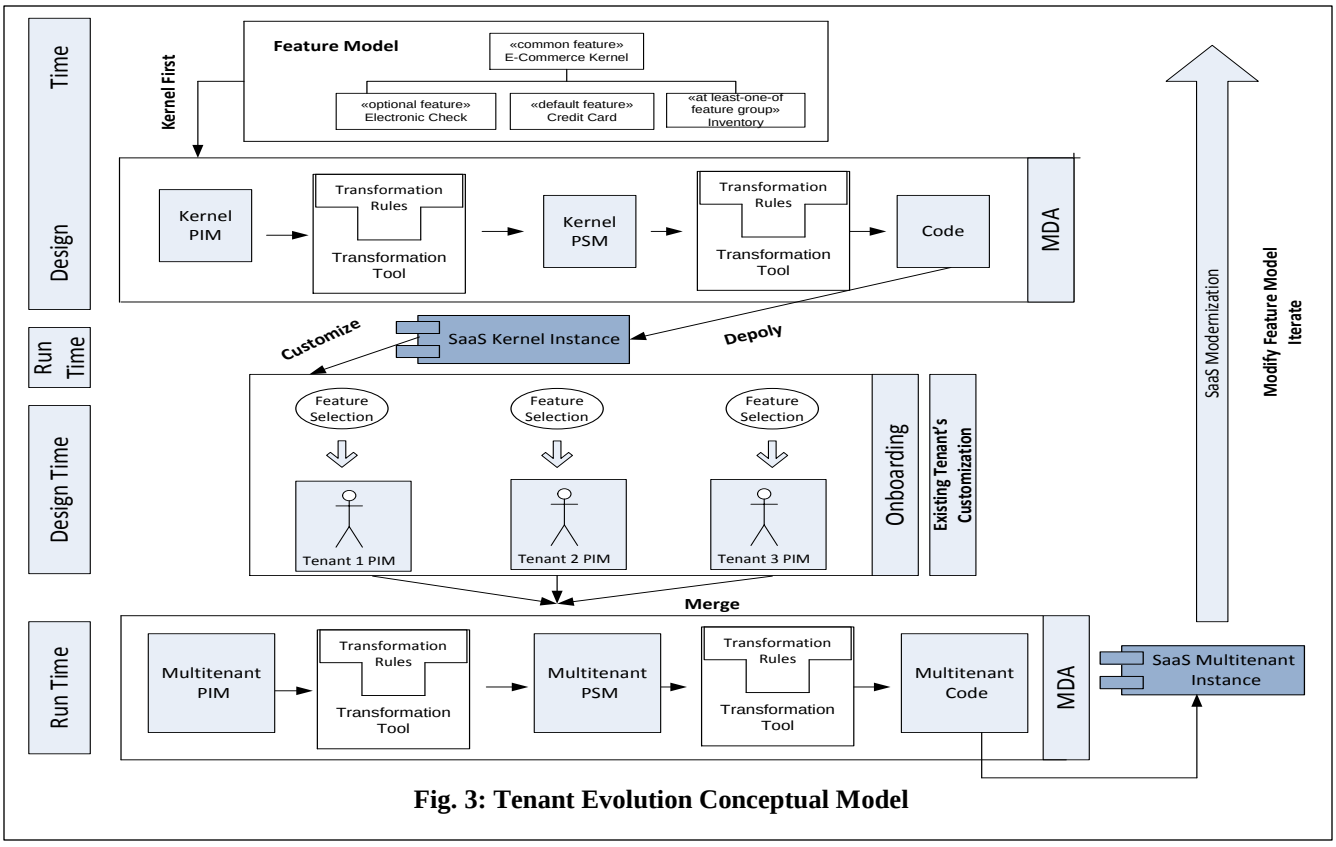

Fig. 3: Tenant Evolution Conceptual Model

performed on the multitenant PIM level before transformation to the PSM.

B. *Onboarding Tenants*

In tenants' onboarding phase, each tenant's application is derived based on the selected features from the feature model similar to the application engineering phase in [5]. Starting with the kernel SaaS, designer will add features to fulfill the requirements of the onboarding tenant. Therefore, each new tenant is represented as a multiple-view Platform Independent Model (PIM) [19]. Once the PIM is transformed to code, new tenants will be deployed by evolving the existing SaaS instance. Deployment of tenants is discussed in the next section.

While customizing an instance, PIMs of the different tenants are merged to generate a multitenant PIM. During the merging, an assessment is performed during which two cases can arise: 1) no tenant configurations' inconsistencies detected, consequently current instance can be adapted and can evolve to answer the tenant new requirements. The multitenant PIM is hence generated and the evolution of the instance will be processed as described above; 2) tenant configurations' inconsistencies detected, consequently current instance cannot evolve to handle the new requirements because of some contradictory requirements. In this case, a management decision should be taken: either a new instance is launched to accommodate the new tenant, or the request is declined as it is deemed undesirable to deploy different instances of a multi-tenant application.

C. *Existing Tenant Customization*

In the existing tenants' customization phase, tenants' customization requests will be handled by evolving the SaaS instance taking into consideration tenant's specific features from the feature model. In a first step, the feature model itself will be evolved with the new customization features before the SaaS is evolved. Then, the PIM of the tenant is updated and used during the merging to generate the multitenant PIM. Again, and similar to onboarding tenants, the requirements change of an existing tenant requires an assessment phase to decide whether a new instance needs to be launched or not.

In the case of a leaving tenant, the multitenant PIM is regenerated from the PIMs of the rest of the tenants and the SaaS code is updated accordingly.

VII. PROOF OF CONCEPT TOOL

As a proof-of-concept, we implemented a tool by extending the Eclipse based SoaSPLE tool [19]. We added the cloud related features (Fig. 1) to SoaSPLE' feature model. In addition, we implemented the cloud related views, i.e. Tenant Configuration, Provisioning, and Platform views and linked them with SoaSPLE's service views. Unlike the Web Services Platform Specific Models (PSM) generated in [19], we use a Service Component Architecture (SCA) [20], and OSGI [21] PSMs instead. SCA and OSGI were used because they facilitate loose coupling and dynamic reconfiguration respectively. We used Apache Tuscany [26] and Apache Felix [27] as the runtime choices for SCA and OSGI respectively.

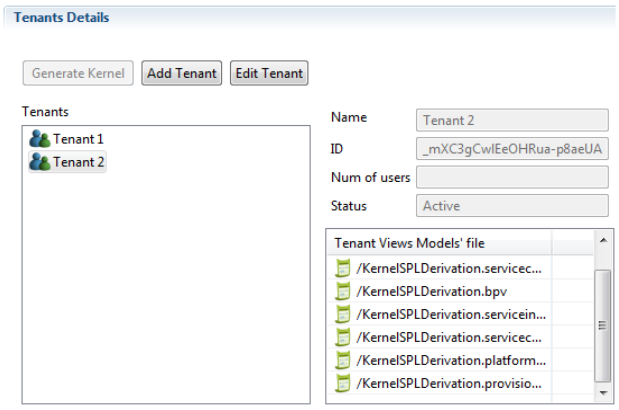

Fig. 4: Phase 1: Kernel Generation

SaaS kernel instance generation is performed by deriving SPL kernel part. Once 'Generate Kernel' button is pressed (Fig. 4), the tool selects all common features of the SPL along with the features' mapped architectural artifacts, i.e. views. The SaaS kernel's PIM is transformed into an SCA Composite PSM. In addition, common features are transformed into OSGI bundles that reside inside the SCA Composite.

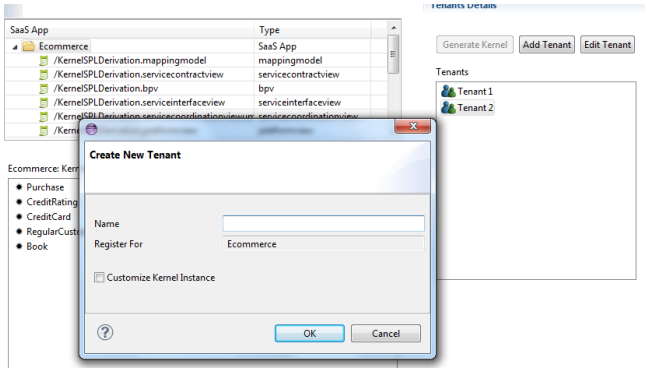

Fig. 5: Phase 2: Onboarding Tenants

Tenant's onboarding is performed by pressing the 'Add Tenant' button. Tenants' specific features are added to the original feature model and highlighted in tenants' designated windows (Fig. 5). Consequently, a tenant's version, i.e. SPL member app, is derived. Each tenant's PIM is transformed into an SCA Composite PSM. In addition, tenant's specific features are transformed into OSGI bundles.

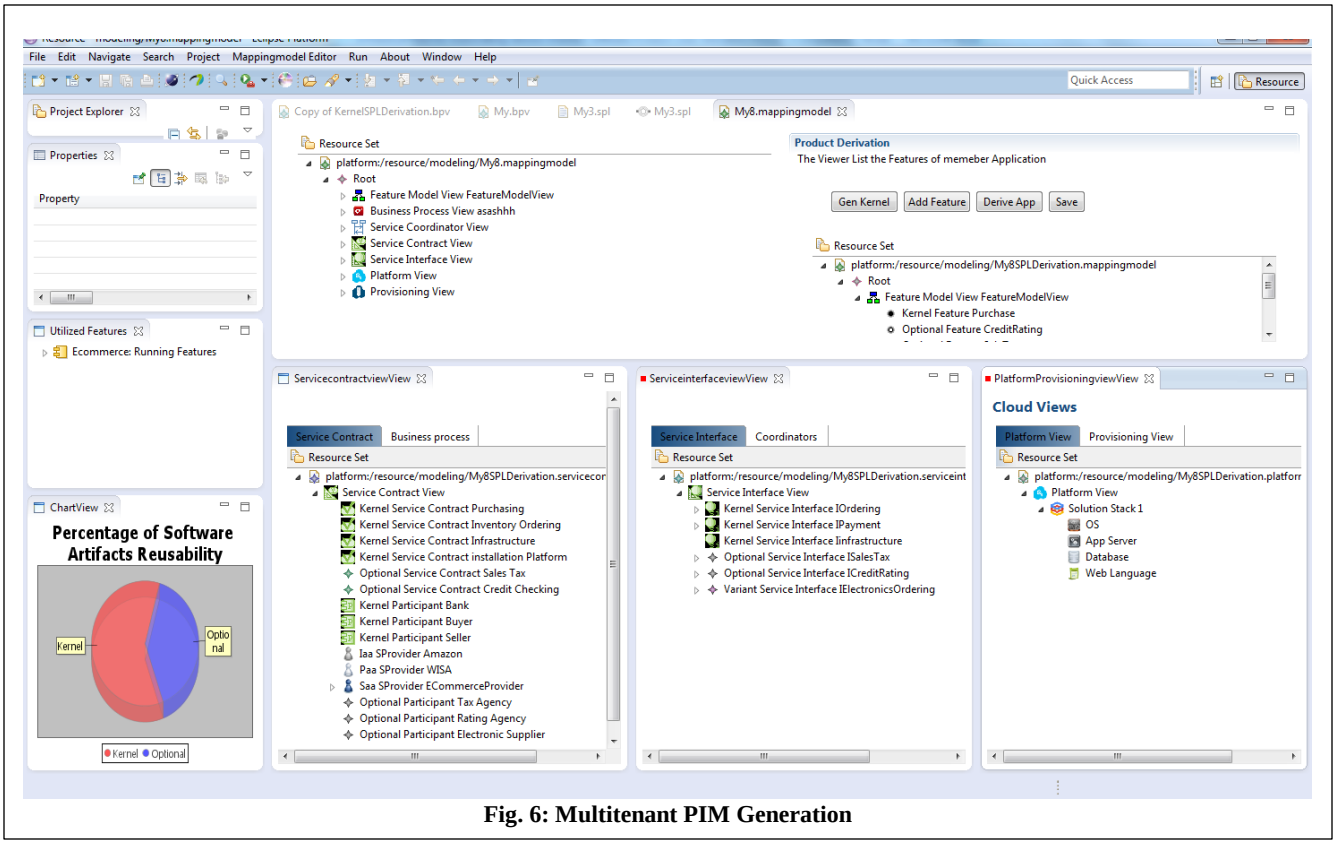

Fig. 6: Multitenant PIM Generation

Existing tenant's customization is handled by pressing the 'Edit Tenant' (Fig. 5) button. Here, a list of existing tenants is displayed where the user can select the tenant in question. Then, customization is handled like in the onboarding step above. Again, added features are mapped to OSGI bundles which will be dynamically added to tenant's SCA Composite.

As tenants features are added or removed, the tool displays a separate Eclipse view, called 'Utilized Features' (Fig. 6), which shows kernel features along with tenants' specific ones. This view acts as a running display of all features currently in the SaaS instance. This view is dynamically updated as tenants' onboarding and customization commence.

An overview of the tool is depicted in Fig. 6 where the different phases and different views are shown.

It should be noted that OSGI bundles are automatically stored in the OSGI registry by the OSGI runtime [21]. Thus, we use the registry to store implemented SaaS features in the form of OSGI bundles. This way, all features from the feature model, once implemented, will be available for use by tenant as prescribed by their corresponding PIMs.

VIII. RELATED WORK

The authors in [11] employ aspect-oriented programming to design service variations for multitenant SaaS systems. SaaS systems are designated as lightweight and heavyweight variants. Heavyweight variants, which need to change their interfaces to cater for variability, are designed using aspects where variability points are modeled as crosscutting concerns. This research only models the interface and implementation views of SaaS, whereas our approach provides a multiple-view modeling coupled with a meta-model that formalizes the aforementioned multiple-view model.

The authors in [12] use SPL and Feature Modeling to create a decision support tool to help cloud providers manage the different application requirements of SaaS systems. Unlike our approach, the approach in [12] is platform-dependent and does not offer multiple-view architectural treatment of cloud services variability.

The authors in [13] discuss customization opportunities for cloud-based SPL. They list advantages and disadvantages of software customization approaches for the three prominent layers of cloud computing, i.e. IaaS, PaaS, and SaaS. While the authors present no specific SPL treatment to realize the customization problem, they merely compare approaches and conclude that PaaS is the most cloud layer that could benefit significantly from SPL approaches.

The authors in [14] argue that because of the intrinsic heterogeneity of cloud based systems, SPL approaches are suitable for cloud-based development. Specifically, feature models are adapted to cater for specific cloud computing concerns like provisioning, customization, and price calculation. The work in [14] is the closest to our work in that it exploits feature models to manage the variability of cloud systems. However, our approach differs in that it uses

feature models as a unifying view to describe variability in multiple views of cloud systems, and provide mapping, based on meta-models, between features and cloud services artifacts.

In line with our work, and to handle variability in multi-tenant SaaS applications, the authors in [25] advocate the need to model the variability at design time as well as run time, and handle the inconsistencies among the requirements of the different tenants. To achieve this goal, they propose to extend the component model (CCM) with tenant configuration and constraints. They also propose to use and extend the Multi-quality auto tuning architecture and THEATRE. However, the applicability of their approach is not validated yet and the tool is still under development. Finally, they do not tackle the problem of how tenant specific customizations can be preserved on the evolution of an MTA, which we are answering in this work using SPL. To the best of our knowledge, there exists no other research that treats variable cloud systems in a multiple-view variability modeling and meta-modeling approach.

IX. CONCLUSION

In this paper, we have introduced a multiple-view modeling approach that addresses service-oriented cloud systems variability concerns in a unified manner. In particular, we have described the integration of SPL concepts of feature modeling and commonality/variability analysis techniques with service modeling approaches (such as SoaML) to model cloud systems variability.

We believe that our approach has several benefits:

- Meta-modeling of the cloud computing ecosystem.
- A multiple-view meta-model for describing cloud-based software product lines.
- Treatment of cloud systems variability concerns in a unified, systematic, platform-independent manner.
- The use of established SPL Feature Modeling to manage variability in the Cloud.

We are currently engaged in the following activities to further our research:

- Modeling elasticity concerns based on the cloud meta-model.
- Enhancing the Coordination view to model the emerging Cloud Service Brokerage (CSB) role.
- Finally, we plan to validate our approach by conducting industrial case studies in healthcare and E-Commerce.

REFERENCES

[1] Peter Mell and Timothy Grance. The NIST Definition of Cloud Computing. Special Publication 800-145, National Institute of Standards and Technology, Bethesda, Maryland, September 2011.

[2] Fang Liu, et al. NIST Cloud Computing Reference Architecture. NIST SP - 500-292, National Institute of Standards and Technology, Bethesda, Maryland, September 2011.

[3] R. N. Taylor, N. Medvidovic, and E. M. Dashofy, Software Architecture: Foundations, Theory, and Practice. Wiley, 2009

[4] H. Gomaa, Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures. Addison-Wesley Professional, 2004.

[5] Abu-Matar, M.; Gomaa, H., "Variability Modeling for Service Oriented Product Line Architectures," Software Product Line Conference (SPLC 2011), , 22-26 Aug. 2011.

[6] Kang, K, S. Cohen, et al. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report CMU/SEI-90-TR-21, Software Engineering Institute, Pittsburgh A, November 1990.

[7] SoaML Beta document, http://www.omg.org/spec/SoaML/1.0/Beta1/, April 2009.

[8] M. Abu-Matar, and H. Gomaa, Service Variability Meta-Modeling for Service-Oriented Architectures,Variability for You Workshop, ACM/IEEE 14th International Conference on Model Driven Engineering Languages and Systems MODELS 2011.

[9] H. Gomaa and M.E. Shin, Multiple-View Modeling and Meta-Modeling of Software Product Lines, Journal of IET Software, Volume 2, Issue 2, Pages 94-122, Published by Institution of Engineering and Technology, April 2008.

[10] Philippe Kruchten, "The 4+1 View Model of Architecture," IEEE Software, vol. 12, no. 6, pp. 42-50, Nov. 1995, doi:10.1109/52.469759.

[11] H. Jegadeesan and S. Balasubramaniam, "A Method to Support Variability of Enterprise Services on the Cloud," in Proceedings of the 2009 IEEE International Conference on Cloud Computing, Washington, DC, USA, 2009, pp. 117–124.

[12] Quinten, C.; Duchien, L.; Heymans, P.; Mouton, S.; Charlier, E.; , "Using Feature Modeling and Automations to select among Cloud Solutions," Product Line Approaches in Software Engineering , 3rd International Workshop on , vol., no., pp.17-20, 4-4 June 2012.

[13] Schmid, K.; Rummler, A., "Cloud-based Software Product Lines," Software Product Line Conference (SPLC), 2012 16th International, vol. 2, pp.164-170, 22-26 Aug. 2012.

[14] Everton Cavalcante, et al. "Exploiting software product lines to develop cloud computing applications," Software Product Line Conference (SPLC), 2012 16th International, vol. 2, pp.179-187, 22-26 Aug. 2012.

[15] SalesForce, http://www.salesforce.com, April 2013.

[16] R. Krebs, C. Momm, and S. Kounev, "Architectural concerns in multi-tenant saas applications," in CLOSER, 2012, pp. 426–43.

[17] K. Schmid and A. Rummler, "Cloud-based software product lines," in Proceedings of the 16th International Software Product Line Conference - Volume 2, New York, NY, USA, 2012, pp. 164–170.

[18] K. Lee, K. C. Kang, and J. Lee, "Concepts and Guidelines of Feature Modeling for Product Line Software Engineering,"

in Proceedings of the 7th International Conference on Software Reuse: Methods, Techniques, and Tools, London, UK, 2002, pp. 62–77.

[19] Abu-Matar, M., Gomaa, H.,"An Automated Framework for Variability Management of Service-Oriented Software Product Lines," Service Oriented System Engineering (SOSE), 2013 IEEE 7th International Symposium on, 25-28 March 2013.

[20] Service Component Architecture (SCA), http://oasis-opencsa.org/sca, May 2013.

[21] OSGI Alliance, http://www.osgi.org/Main/HomePage, May 2013.

[22] A. Kleppe, J. Warmer, and W. Bast, MDA Explained: The Model Driven Architecture(TM): Practice and Promise, 1st ed. Addison-Wesley Professional, 2003.

[23] D. S. Frankel, Model Driven Architecture: Applying MDA to Enterprise Computing. Wiley, 2003.

[24] Klaus Schmid and Andreas Rummler. “Cloud-based Software Product Lines”. In: Proceedings of the 2nd International Workshop on Services, Clouds, and Alternative Design Strategies for Variant-Rich Software Systems (SCArVeS 2012) at the 16th International Software Product Line Conference (SPLC’12). Vol. 2. ACM, 2012, pp. 164–170.

[25] Julia Schroeter, Sebastian Cech, Sebastian Götz, Claas Wilke, and Uwe Aßmann. 2012. Towards modeling a variable architecture for multi-tenant SaaS-applications. In Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems(VaMoS '12). ACM, New York, NY, USA, 111-120.

[26] Apache Tuscany, http://tuscany.apache.org/, October 2013.

[27] Apache Felix, http://felix.apache.org/, October 2013.