SaaS Dynamic Evolution Based on Model-Driven Software Product Lines

Article · February 2015

DOI: 10.1109/CloudCom.2014.131

Fatma Mohamed
Khalifa University
12 PUBLICATIONS **51** CITATIONS
SEE PROFILE

Mohammad Abu Matar
The University of Arizona
25 PUBLICATIONS **253** CITATIONS
SEE PROFILE

Rabeb Mizouni
Khalifa University
127 PUBLICATIONS **1,148** CITATIONS
SEE PROFILE

Mahmoud Al-Qutayri
Khalifa University
265 PUBLICATIONS **2,715** CITATIONS
SEE PROFILE

SaaS Dynamic Evolution Based on Model-Driven Software Product Lines

Fatma Mohamed
Khalifa University
Abu Dhabi, UAE
fatma.mohamed@kustar.ac.ae

Mohammad Abu-Matar
Eitsalat British Telecom Innovation Center
Abu Dhabi, UAE
mohammad.abu-matar@kustar.ac.ae

Rabeb Mizouni
Khalifa University
Abu Dhabi, UAE
rabeb.mizouni@kustar.ac.ae

Mahmoud Al-Qutayri
Khalifa University
Abu Dhabi, UAE
mqutayri@kustar.ac.ae

Zaid Al Mahmoud
Khalifa University
Abu Dhabi, UAE
zaid.almahmoud@kustar.ac.ae

Abstract— **Cloud computing is an emerging paradigm that provides scalable computing and storage capabilities where resources are accessed on a pay-as-you-go basis. Software as a Service (SaaS) applications are hosted in the cloud and made available as services for tenants' organizations over a network. To achieve reusability in cloud computing, software and hardware resources are shared among multiple tenants. Conventional multi-tenant SaaS applications provide the same set of services for all tenants thus resulting in one-size-fits-all applications. However, as tenants may have different requirements, customizable SaaS solutions are needed. To accommodate evolving tenants' requirements, the SaaS instance should evolve systematically. In this paper, we present a multi-tenant single instance SaaS evolution platform based on Software Product Lines (SPLs). The platform specifies a set of evolution rules, based on feature modeling, that govern evolution decisions. We also present the early implementation phases of the proposed approach based on SPLs and Model Driven Architecture (MDA) concepts.**

Keywords—saas; dynamic evolution; software product lines; model driven engineering; multi-tenant applications

I. Introduction

Recent advances in information and communication technology (ICT) have shifted the use of conventional licensed-based software to widely accessible cloud-based applications [1]. Software-as-a-Service (SaaS) has recently been adopted by many organizations, tenants, to get their work done through subscription-based services. To leverage economies of scale, SaaS applications are provided to multiple tenants sharing the same software and hardware resources based on virtualization concepts [2]. This can be referred to as multi-tenant applications (MTAs). Because tenants subscribing to a SaaS application have varying requirements, a one-size-fits-all SaaS that provides the same set of services for all tenants is not an efficient solution. Customizable versions of the SaaS application can be delivered in accordance with tenants' needs based on software product lines (SPLs) principles.

An SPL [3] is a family of software products that share a set of common features while exhibiting some variations distinguishing each single product. Software product line engineering (SPLE) is the discipline that supports modeling commonality and variability among SPL variants. SPLE emphasizes variability management (VM) as a key principle for developing family of applications and it is used to specify the dependencies among the set of common and varying features to support their instantiations process. Feature models [4] are the most commonly used technique to capture SPL's commonality and variability at the requirements and design stages. A feature diagram is a graphical representation of nodes and associations that represents the common and variable features of an SPL.

Following this concept, each member of the SaaS family shares significant commonalities while exhibiting some variations. Providing a customizable single instance multi-tenant (SIMT) SaaS application is a challenging problem when it comes to seamless system evolution. Maintaining a single instance per multiple tenants increases the management burdens on the service providers in order to deliver the right part of the application to the right tenant based on their specific requirements. To address this challenge, we propose a feature-oriented framework that raises the variability management abstraction level to facilitate the dynamic evolution of SIMT SaaS applications based on service-oriented SPLs and Model Driven Architecture (MDA).

MDA [5] is an approach adopted by the Object Management Group (OMG) to provide reusable solutions addressing the whole development lifecycle. MDA relies on the use of models, with various abstraction levels, to carry out the different phases of system development. MDA captures the system requirements independently from any platform specific details using a platform-independent model (PIM). Then, it integrates the platform-specifications to a PIM to generate a platform-specific model (PSM). Model-to-Model (M2M) transformations are used to transform one model to another based on a set of predefined rules.

Our research aims to support the dynamic evolution of SIMT SaaS applications based on model-driven SPLs. MDA adds an automation layer to our solution where multi-view M2M transformations, embedding a set of evolution rules, are used to carry out the evolution process independently from any platform specific details. A set of consistency checking rules are defined and formulated in the transformations to guarantee the consistency of the consecutively generated models and to maintain the mappings between the problem space and the solution space artifacts.

The contribution of this paper is threefold: (1) Specifying a framework that supports the dynamic evolution of SIMT SaaS applications. (2) Defining a set of evolution rules to govern the evolution decisions (3) Developing the infrastructure to support runtime evolutions. (4) Presenting a proof of concept tool.

The rest of the paper is organized as follows: Section 2 elaborates on the problem statement. Section 3 describes the SaaS dynamic evolution process model. Section 4 provides the implementation details of the early phases illustrated by an example. Section 5 presents a proof of concept tool. Section 6 discusses the related work and Section 7 concludes the paper.

II. PROPLEM ELABORATION

A conventional SaaS MTA provides the same functionalities for all tenants subscribing to it. The concept of multi-tenancy was introduced to leverage economies of scale by sharing software and hardware resources [6]. However, this imposed some restrictions on the nature of the application being shared among tenants. Tenants who are subscribing to the same SaaS instance are provided with the same set of services. In reality, different tenants may have different requirements for the same application. Thus, customizable SaaS versions should be provided to satisfy the specific requirements of each tenant. In the current state-of-the-art, this can be achieved using SPL concepts, as in [7] and [8]. In our work, a SIMT SaaS can be customized at runtime to satisfy the changing requirements of the subscribing tenants. This is different than conventional Dynamic Software Product Lines (DSPLs) where each tenant has a single instance that can be customized at runtime [9]. The challenge that we are tackling in our approach is to evolve a single instance, dynamically, for multiple tenants and guide the evolution process using a rule-based engine to preserve the consistency of the evolved instance.

Commonality and variability among the whole set of product variants, i.e. tenants, are captured using feature models [10]. Then, a tenant-specific PIM, which will be discussed in the next section, is derived during application engineering by integrating the common features with the various functionalities distinguishing each tenant.

There are several challenges in developing a customizable SIMT SaaS applications – most importantly, evolving the SaaS instance to satisfy the tenants' changing requirements. As any other software system, the evolution of a SaaS application necessitates its unavailability for, at least, a short time. Dynamic adaptation techniques [11] have been introduced to support run-time evolutions without compromising the operational integrity of the system. However, the system's status should be validated to guarantee that the proposed evolution takes the system from its initial consistent state to another consistent state that satisfies the desirable needs. Considering SaaS applications, the challenge would be: how to support and automate the dynamic evolution of SIMT SaaS applications without compromising neither the consistency of the up-running instance nor the operational integrity of the subscribing tenants? This is, in essence, different than the single tenant per instance paradigm. In fact, in MTAs multiple tenants share one code base and have access to the same resources. Evolution in this case should be tenant-aware in which customization requests are reflected only on the requesting tenant without effecting the rest.

III. SAAS DYNAMIC EVOLUTION FRAMEWORK

In previous work [12] we have introduced a process model to support multi-view runtime evolutions of SIMT SaaS applications. This paper realizes the suggested model by specifying a framework with a set of evolution rules and model-to-model transformations to govern the evolution process. This section provides an overview of the evolution framework.

Our multi-view SaaS evolution framework is based on a multi-view cloud variability meta-model that was defined in [12] and [13]. The cloud multi-views meta-model integrates eight meta-views: (1) Feature meta-view that represents the feature variability among the SaaS family. (2) Service contract meta-view that identifies the SaaS participants. (3) Business process meta-view that determines the workflow of tasks. (4) Service interface meta-view that describes the operations of the services. (5) Provisioning meta-view that defines the infrastructure needed whenever Infrastructure-as-a-Service (IaaS) is provided. (6) Platform meta-view that defines the variability regarding the web languages, databases, servers and operating systems. (7) Service coordinator meta-view that coordinates the invocation of services, and finally a (8) Tenant configuration meta-view that keeps track of all tenants' configurations.

The evolution framework, Figure 1, comprises three iterative phases: Developing a SaaS kernel, Handling onboarding tenants and Customizing existing tenants. During the SaaS kernel development phase, the common features are selected from the SPL feature model and represented as a kernel PIM. The kernel PIM is considered as the base of constructing tenants' configurations.

Onboarding tenants identify their needs during a feature selection phase. The set of selected features is then integrated with the kernel PIM to generate a tenant-specific PIM. Consequently, all tenants' specific PIMs are merged to generate a multi-tenant PIM satisfying the requirements of all the subscribing tenants.

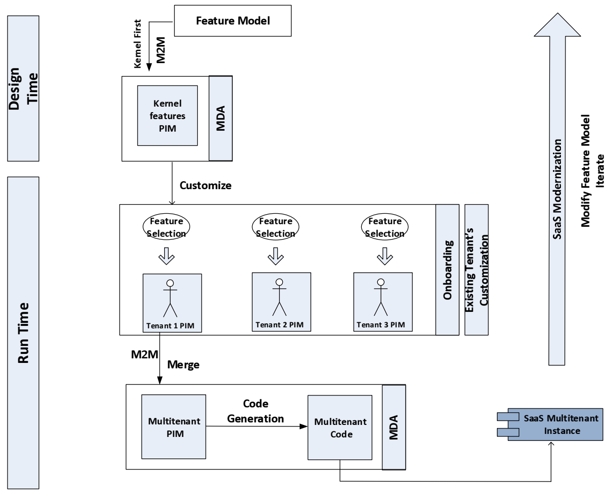


Fig 1. SaaS Dynamic Evolution Conceptual Model

An assessment, which will be discussed in detail in section IV, is performed to ensure the consistency of the multi-tenant PIM and to facilitate the SaaS evolution decisions.

Code generators are used to generate a multi-tenant platform specific code given the multi-tenant PIM. The generated code represents a layer that integrates with a dynamic, component-based runtime target environment.

Existing tenants may customize their configurations, based on a feature re-selection phase, causing their corresponding tenant-specific PIMs to be updated. This implies that all tenant-specific PIMs are remerged to guarantee that the new tenant configuration doesn't violate the consistency of the SaaS instance before it evolves to satisfy the customization request. In general, customization requests consist of the tenant identifier, the newly selected features and the current status of the instance.

IV. SaaS Evolution Platform Implementation

The platform early development process was carried out by: (1) Specifying the possible evolution scenarios (2) Defining a set of dynamic evolution rules (3) Implementing m2m MDA transformations and, finally, (4) Developing a proof of concept.

A running example is introduced below followed by a detailed description of the previously mentioned activities.

A. A Running Example

To illustrate our framework, we present a simple example of an e-commerce SaaS application. Its feature model is represented using FODA notation [10] and consists of several common, i.e. kernel, alternative, and optional features.

B. SaaS Evolution Scenarios

This section describes the different scenarios where a SaaS instance evolution could be required. It also presents the rules (shown in Table 1) that have been developed to govern SaaS dynamic evolution decisions. See Figure 2.

Scenario 1: Instantiating a SaaS kernel. In this scenario, the feature model is traversed for kernel features to generate a kernel PIM. The kernel PIM consists of the core features shared among all tenants.

In the E-commerce example, the kernel PIM consists of *Fulfillment*, *Purchasing*, *Ordering* and the defaults of the kernel features including *Regular Consumer*, *Books* and *Electronic Check* along with their mappings to the other views. These mappings integrates the service interfaces, work flows, data models and the other views with the kernel features.

Scenario 2: Onboarding Tenants. In this scenario, a new tenant initiates a request to subscribe to the SaaS services. Assuming that the functionalities to be requested by the tenant are part of the existing feature model, a tenant starts by selecting the desired feature from the feature model. Then, for each selected feature, the tenant picks the preferred interfaces workflows and data models, if available.

A feature mapping is a link that is created automatically between a feature in the feature model and the elements of the multi-views based on the tenant's preferences. For example, a tenant can choose among available service interfaces, business processes and many other elements whenever there are more than one choice provided by the service provider. This occurs at the feature selection stage where a tenant is provided by the features and their different mappings to select from.

A feature can also be associated with a data model varying between a *shared-database shared-schema*, a *shared-database separate-schema*, and a *separate database*. Using MDA transformations, the same feature subscribed to by different tenants can be mapped to different data models based on the tenant specific requirements. As for *Payment for* example, some tenants may approve storing their information in a shared database with separate-schema. Others, who are concerned about security, may request a separate database to keep their data fully isolated. Generally, different choices for feature mappings and data models are reflected in the subscription cost. It is worth mentioning that there are two types of data stores associated with our approach. A *SaaS data store* that is used to maintain tenants' configurations, mappings and so on, and *tenants' data stores* that maintains tenants' private data.

The set of selected features, their mappings, and the associated data model is called a tenant configuration. The tenant's configuration along with the tenant's ID and the current status of the instance forms the subscription request. The status of the system is represented as a flag, Boolean value, that indicates whether the multi-tenant instance is consistent or not according to the rules specified in Table 1. Once the request is received by the platform, an assessment is performed to ensure the feasibility of the tenant's configuration according to *Rule 1* in Table 1.

The feature-level consistency checking rules ensure the validity of the tenant configuration used to generate a valid SPL member later on; namely a tenant-specific PIM. If the consistency rules reveal some contradictions, the tenant's request is rejected since the SaaS instance can't be evolved with the presence of inconsistencies.

After ensuring the validity of the tenant's configuration, the kernel PIM is integrated with the tenant configuration to generate a tenant-specific PIM.

A tenant-specific PIM is a valid SPL member that is represented as a set of features that combines the common features shared among all tenants and the variable features selected by a specific tenant, along with the feature mappings.

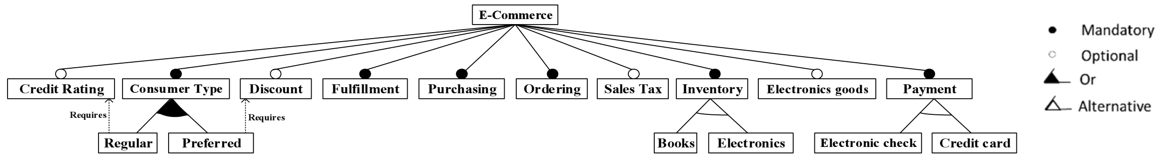


Fig 2. E-Commerce Feature Model

TABLE I. DYNAMIC EVOLUTION RULES

Rule no.	**Description**
Rule 1	A feature requested by an onboarding tenant is added to the tenant-specific configuration if and only if the feature-level consistency of the tenant-specific configuration can be maintained. • Consistency: The tenant-specific configuration complies with the feature model constraints according to the following sub-rules: - At least one feature should be selected from an OR feature group. - One and only one feature should be selected from Alternative feature group. - Zero or more features should be selected from an Optional feature group. - If a selected feature requires any other feature, the platform automatically adds the features required. - Exclusive features can't be selected at the same time.
Rule 2	Committing changes to a valid tenant configuration entails recording the tenant configuration, generating a tenant-specific PIM and merging the selected features to the multi-tenant PIM. • Recording: Keeping track of the tenant-specific configurations using in a database. Generating & Merging: This is done by MDA M2M transformations.
Rule 3	Generating a tenant-specific PIM necessitates mapping the features of the tenant configuration to the architectural artifacts in the multiple views. • Mapping: Recording the traceability links between the feature and the multi-views to a database.
Rule 4	A feature is not added to the multi-tenant PIM unless it is not part of any other existing tenant's-specific configuration. • Added: The feature becomes part of the up-running instance; this includes deploying a new set of services that corresponds to the feature and feature mappings. If the feature already exists in the multi-tenant PIM, check feature mappings and data models; if Rule 10 is satisfied or new feature mappings are detected, the new feature mappings or data models are added, cumulatively, to the feature that exists in the multi-tenant PIM and the SaaS instance evolves, otherwise, no evolution is required.
Rule 5	Adding a new feature that is part of the existing feature model to the multi-tenant instance entails evolution. • New Feature: A feature that has not been selected before by any of the existing tenants. • Evolution: includes re-constructing the business process model, constructing/updating data models, binding to service interfaces and re-deploying the instance.
Rule 6	Adding a new feature to the multi-tenant configuration necessitates adding its associated artifacts to the other views. • New Feature: A feature that has not been selected before by any of the existing tenants. • Fetching: Accessing the service line database, getting the feature's multi-views' artifacts and adding them to the instance.
Rule 7	The multi-tenant instance can be evolved if and only if all of the following are satisfied: - All tenants' configurations are consistent in the feature-level; this means that every tenant configuration complies with the feature model constraints. - The interacting services are interoperable; this means that the services can be integrated together smoothly in one instance without generating unexpected behaviors. - The requested customization is deployed first to a testing environment to identify the changes' impact. - The current status of the instance allows evolution; this means that the instance is consistent, available and scalable. - No errors, failures or overflows are detected in the queues and data stores of the instance. - Third party components are verified to be tenant-aware and interoperable.
Rule 8	The instant is not evolved and the changes are rolled back if and only if one of conditions specified in Rule 7 is unsatisfied.
Rule 9	Requesting a new feature that is not part of the existing feature model requires evolving the feature model. • Evolving the feature model includes: - Injecting the newly requested feature to its proper place in the feature model according to its relevance to the existing features. - Imposing feature's constraints. - Revalidating the feature model.
Rule 10	Selecting a new shared-database separate-schema or separate database data model for an existing feature entails evolution.
Rule 11	If an onboarding tenant requests to have a shared-database shared-schema data model with the selected feature then, the existing schema is used to insert new records indexed by TenantID.
Rule 12	If an onboarding tenant requests to have a shared-database separate-schema data model with the selected feature then, a new tenant-specific schema is created and the requesting tenant is authorized.
Rule 13	If an onboarding tenant requests to have a separate-database data model with the selected feature then, a new tenant-specific database is created and the requesting tenant is authorized.
Rule 14	A tenant's subscription/customization/leaving request not necessarily entails evolution. • Customization request: May include adding a feature, removing an existing feature, modifying business process, switching to another service interface, switching to another data model, upgrading to more reliable services and more. • Not necessarily: If the current instance contains the selected feature, business process, service interface or the services requested by the tenant, new traceability links are injected to associate the tenant with the requested element without the need to evolve the instance; or if the features, the feature mappings and the data models of a leaving tenants are utilized by other existing tenants.
Rule 15	An existing feature can't be removed from the multi-tenant instance unless it is not part of any other existing tenant's configuration. • Removed: Disabling the feature from the multi-tenant instance.
Rule 16	Removing feature from the multi-tenant instance necessitates removing its mapping to the other views. • Removing its mapping: Includes accessing the service line database, getting the associated views and disabling them.
Rule 17	Removing features from the multi-tenant instance entails evolution. • Evolution: includes disabling the associated business process model, service interfaces and un-deploy the selected feature.
Rule 18	A feature should comply with different data models. • Comply: A feature should support different implementations of data models, this includes the different data model structures and query languages.
Rule 19	Data services of the instance cannot access/modify tenant's data unless it has data access/modification permissions.
Rule 20	To access/manipulate features or data, a tenant should be authorized.

The integration is done via a M2M transformation that accepts the tenant configuration and the kernel PIM as an input and generates a tenant-specific PIM as an output. After that, the newly generated tenant specific PIM is merged with the current multi-tenant PIM, that contains all features and feature mappings of all subscribing tenants, to accommodate the new requirements. This phase is done according to *Rules 2* and *3* in Table 1.

In the E-commerce example, let us assume that **Tenant 1** selects the following features: *Sales Tax*, *Books*, *Electronic Check* and *Preferred Customer*. Applying the consistency checking rules, the platform will add the feature *Discount* automatically to the configuration since it is required by *Preferred Customer* feature (Figure 3a). The tenant-specific PIM will consist of: *Fulfillment*, *Purchasing*, *Ordering*, *Sales Tax*, *Books*, *Electronic Check*, *Preferred Customer,* and *Discount* features along with their mapping to the other views. At the same time, an onboarding tenant, **Tenant 2**, selects the desired features as shown in Figure 3b. Assuming that **Tenants 1** and **2** are the only tenants subscribing to the E-commerce SaaS, their tenant-specific PIMs are merged to get a Multi-tenant PIM. This merging is, in fact, done according to *Rules 4, 5* and *6* shown in Table 1.The resulting multi-tenant PIM is shown in Figure 3c.

If a tenant request results in updating the multi-tenant PIM by adding new features, applying *Rules 4,5,6,10,11,12* and *13* in Table 1, code generators are used to get the corresponding multi-tenant code. Code generators are simply MDA transformations that map a model to specific technologies. Thus, they can be considered as one to many mappers that are based on MDA transformations. Finally, the SaaS instance evolves according to *Rule 7* to accommodate the new requirements.

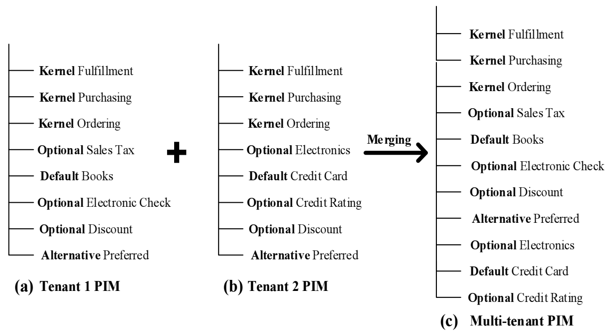


Fig 3. Generating a Multi-Tenant PIM

Onboarding tenants do not necessarily require evolving the SaaS instance. If all features and feature mappings requested by an onboarding tenant are already provided by the existing multi-tenant instance, according to *Rule 14*, new traceability links are created, automatically, to associates the tenant with the requested feature without evolving the SaaS instance.

Another case that should be considered is onboarding tenants with new requirements that are not part of the SaaS feature model. In this case, a feature model evolution is required to integrate the new features requested by a tenant according to *Rule 9* in Table 1. This will depend on two important factors: the relevance of the new requirements to the existing ones and the number of tenants requesting it. Again, the tenant's request can be rejected if the SaaS instance can't be evolved to accommodate the new requirements. This case will be considered for further investigation during the next phase of the research work.

Scenario 3: Customizing Existing Tenants. Since SaaS services are accessed on a pay-as-you-go basis, existing tenants may request to customize their own feature configurations based on their changeable requirements over time. Through a feature reselection phase, a tenant selects the set of features to be customized. A tenant may request to add/remove features, change feature mappings or change feature's data model. Selecting a new data model for an existing feature may entail evolution according to *Rules 10-13*.

A customization request results in automatically updating the tenant-specific PIM by accessing the application's data store and fetching the feature mappings and update them based on the added/removed/updated features. Consequently, consistency checking rules are applied to ensure the feasibility of the newly selected configuration according to *Rule 1*.

After updating the tenant-specific PIM, the multi-tenant PIM is also updated by merging the existing tenants' specific PIMs with the updated tenant-specific PIM. An assessment is performed according to *Rules 4, 5, 6, 15, 16* and *17* in Table 1. If *Rule 14*, in Table 1, is satisfied, the SaaS instance can evolve. Otherwise the changes are rolled back according to *Rule 8*. Rolling back changes is necessary when unexpected behaviors/errors happen during evolution. For example, a service failure, during the evolution process, necessitates rolling back the system to its initial consistent state before evolution. This helps preserving the consistency of the multi-tenant instance.

If *Rule 14*, in Table 1, is satisfied, the customization request will be processed without evolving the SaaS instance.

Scenario 4: Removing Tenants. As the cost model of SaaS applications suggests, tenants can subscribe/unsubscribe to/from services based on their needs. At a given time, a tenant may decide to unsubscribe from all the services they requested before. Depending on the leaving tenant's configuration, a SaaS evolution may or may not be needed to remove the tenant. First, the corresponding tenant-specific PIM is checked against the multi-tenant PIM according to *Rules 15* and *16* in Table 1.

When *Rule 15* is satisfied, the multi-tenant PIM is updated to accommodate the features utilized only by the up-running tenants. The leaving tenant-specific PIM is then removed, the updated multi-tenant PIM is transformed into code and the SaaS instance evolves according to *Rules 7* and *17*. Again, removing a tenant may not require evolving the SaaS instance. According to *Rule 15*, in Table 1, if all features of the leaving tenant are subscribed to by other tenants, the tenant is removed without evolving the instance.

For all the specified rules, an evolution of a SaaS instance is done only once after entirely updating the multi-tenant PIM. When removing a tenant for example, the SaaS instance evolves only once by removing all the features that are exclusively used by the leaving tenant.

A leaving request may be rejected if at least one of the tenant's services is not in a consistent state. This means that some of service's transactions are not finalized yet. Such information is fetched from the underlying component-based environment that is integrated with the multi-tenant code to reflect the dynamic evolution on component-level.

C. *Early Implementation Phases*

In previous work [13], we have constructed the meta-models of the feature model and the multi-views, instantiated them and used them to develop a tool that implements the feature selection module. Eclipse Modeling Framework (EMF[1]) was used to define the meta-models as ECORE files. Constraints on meta-models were specified using Object Constraint Language (OCL[2]), which is a formal language used to define conditions on model objects. OCL constraints were applied to specify the static feature-level consistency checking rules to the feature meta-model to ensure the consistency of the selected tenants' configurations during feature selection.

The feature meta-model is based on FODA notation to describe features and feature relations [10]. Each feature in the feature model was mapped to the different views. The mapping is maintained during all of the transformations carried out by the evolution process.

In this work, a multi-tenant PIM is constructed through the following phases:

Phase 1: Generating a kernel PIM. In this phase a M2M transformation is applied to generate a kernel PIM from a given feature model. All M2M transformation carried out by the framework are based on Query-View-Transformation (QVT[3]) which is a standard language adopted by the OMG and used to carry out transformations between models.

Phase 2: Constructing a tenant-specific PIM. This is done via another M2M transformation where the inputs are the kernel PIM, the selected configuration and the tenant ID. Throughout the transformation, the features and feature mappings forming the tenant configurations are integrated with the SaaS kernel PIM to generate a tenant-specific PIM. This phase is repeated for each new tenant requesting to subscribe to the SaaS services.

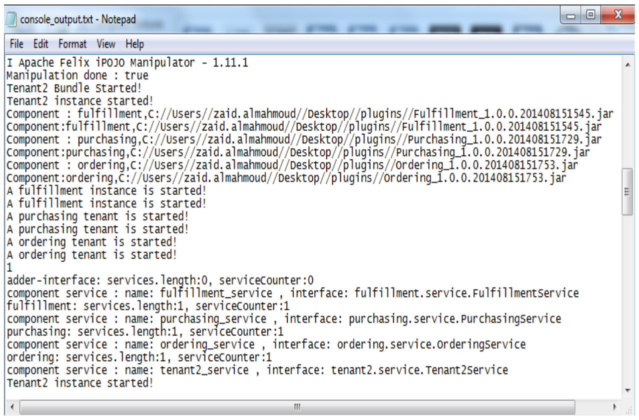
Phase 3: Generating a multi-tenant PIM. In this phase, the existing tenant-specific PIMs and a newly added/updated tenant-specific PIM are merged together according to the rules discussed in the previous section. The applied transformation results in constructing a multi-tenant PIM that consists of all features and features' mappings utilized by all tenants. All the transformations have been implemented in QVT and tested using Eclipse EMF.

Figure 4 shows a QVT code snippet that carries a M2M transformation based on Rules 4 and 6 shown in Table1.

```
1. mapping Feature:: generate_multiTenantPIM() {
2. tenant_Set:= newTenantPIM.objects()[Feature].name->asOrderedSet();
3. multi_tenant_Set:= currMultiTenantPIM.objects()[Feature].name->asOrderedSet();
4. intersection_Set:= tenant_Set->intersection(multi_tenant_Set);
5. if (intersection_Set->includes(self.name))then{
6. log ("This feature already exists in the multi-tenant PIM.");
7. config_Set->select(w|w.name=self.name)->at(1).tenantsList:=config_Set->
8. select(w|w.name=self.name)-> at(1).tenantsList+ newTenantID;
9. }else{
10. new_multiTenantSet:= new_multiTenantSet ->append(object Feature
11.  {name:=self.name;
12. id:=self.id;
13. dependentOn:=self.dependentOn;
14. reuseStereotype:=self.reuseStereotype;
15. supportedByContract:= self.supportedByContract;
16. supportedByParticipant:=self.supportedByParticipant;
17. supportedByBusinessProcess:=self.supportedByBusinessProcess;
18. supportedByServiceInterface:=self.supportedByServiceInterface;
19. supportedByCoordinator:=self.supportedByCoordinator;
20. supportedByMessage:=self.supportedByMessage;
21. supportedByTask:=self.supportedByTask;
22. supportedByInfrastructure:=self.supportedByInfrastructure;
23. supportedByCompositStructure:=self.supportedByCompositStructure;};}endif;}...
```

Fig 4. QVT Code Snippet

The inputs of the transformation are a new/updated tenant-specific PIM and the current multi-tenant PIM. Features that belong to the intersection set of both PIMs are not re-added to the multi-tenant PIM, rather, the platform ensures that these features are referenced by the requesting tenant (lines 5-8).



```
I Apache Felix iPOJO Manipulator - 1.11.1
Manipulation done : true
Tenant2 Bundle Started!
Tenant2 instance started!
Component : fulfillment,C://Users//zaid.almahmoud//Desktop//plugins//Fulfillment_1.0.0.201408151545.jar
Component:fulfillment,C://Users//zaid.almahmoud//Desktop//plugins//Fulfillment_1.0.0.201408151545.jar
Component : purchasing,C://Users//zaid.almahmoud//Desktop//plugins//Purchasing_1.0.0.201408151729.jar
Component:purchasing,C://Users//zaid.almahmoud//Desktop//plugins//Purchasing_1.0.0.201408151729.jar
Component : ordering,C://Users//zaid.almahmoud//Desktop//plugins//Ordering_1.0.0.201408151753.jar
Component:ordering,C://Users//zaid.almahmoud//Desktop//plugins//Ordering_1.0.0.201408151753.jar
A fulfillment instance is started!
A fulfillment instance is started!
A purchasing tenant is started!
A purchasing tenant is started!
A ordering tenant is started!
A ordering tenant is started!
1
adder-interface: services.length:0, serviceCounter:0
component service : name: fulfillment_service , interface: fulfillment.service.FulfillmentService
fulfillment: services.length:1, serviceCounter:1
component service : name: purchasing_service , interface: purchasing.service.PurchasingService
purchasing: services.length:1, serviceCounter:1
component service : name: ordering_service , interface: ordering.service.OrderingService
ordering: services.length:1, serviceCounter:1
component service : name: tenant2_service , interface: tenant2.service.Tenant2Service
Tenant2 instance started!
```

Fig 5. Runtime Environment Integration

The remaining features that are not part of any other existing tenant, and their mappings to the other views are added to the multi-tenant PIM (lines 9 to 23).

The PIMs generated by the platform are represented as XMI[4] files. An XMI file consists of nested tags that correspond to features and feature mappings.

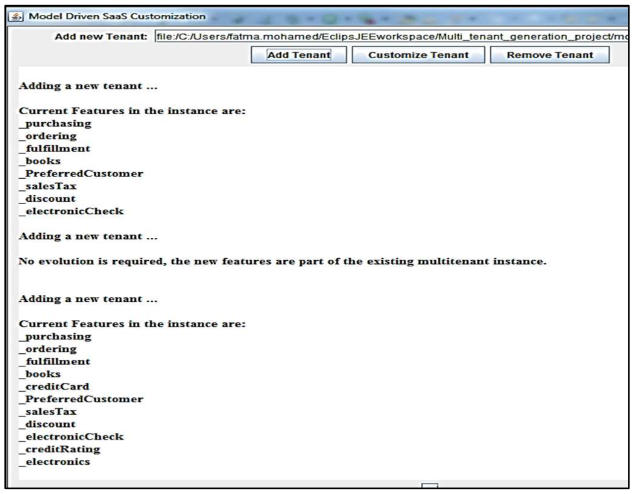
To support run time, the platform can be integrated with component-mapping tools that accepts XMI files as input. The multi-tenant PIM and the component's mapping file are used to specify the type and the number of components shared by all the subscribing tenants; while tenant-specific PIMs specifies the number of component instances needed to serve individual tenants. Each time a PIM is updated, the changes will be given

[1] eclipse.org/modeling/emf
[2] http://www.omg.org/spec/OCL
[3] http://www.omg.org/spec/QVT
[4] http://www.omg.org/spec/XMI

as an input to the tool. The tool manages components accordingly.

V. Proof of Concept Tool

As a proof of concept, we have implemented a tool using the Eclipse plugin development environment.



```
file:/C:/Users/fatma.mohamed/EclipsJEEworkspace/Multi_tenant_generation_project/mo
```

```
Adding a new tenant ...

Current Features in the instance are:
_purchasing
_ordering
_fulfillment
_books
_PreferredCustomer
_salesTax
_discount
_electronicCheck

Adding a new tenant ...

No evolution is required, the new features are part of the existing multitenant instance.


Adding a new tenant ...

Current Features in the instance are:
_purchasing
_ordering
_fulfillment
_books
_creditCard
_PreferredCustomer
_salesTax
_discount
_electronicCheck
_creditRating
_electronics
```

Fig 6. Adding a New Tenant to the Multi-tenant PIM

The plugin is responsible for executing the QVT transformations that were defined based on the evolution rules. *Rules 1-6* and *15-17* have been implemented inside the transformation as OCL, and supported outside using Java, so that evolution decisions can be made before, during, or after executing the transformations based on the provided conditions. Using this tool you can add, remove, and customize tenants, display the current features of the multi-tenant instance and observe the changes caused by executing model transformations.

Figure 6 shows adding three onboarding tenants to the multi-tenant instance given that the second addition was for a tenant configuration containing a subset of the features added for the first tenant. No evolution is required since the multi-tenant instance has all the features requested by the second tenant.

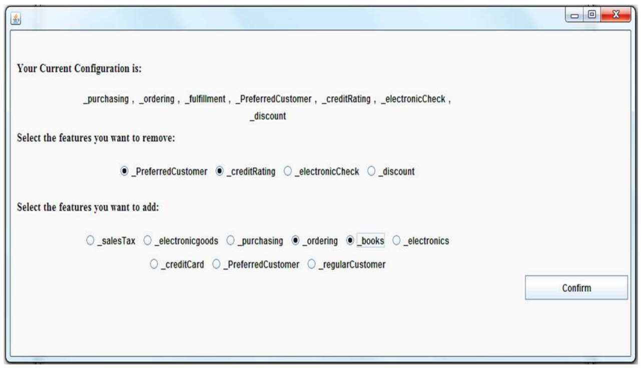
In Figure 7, a tenant requests to customize his own configuration by selecting/deselecting the desired features. After the tenant confirms the new configuration, the updated tenant-specific PIM is merged with the multi-tenant PIM.

The tool output, which represents a multi-tenant PIM, is an XMI file that contains the features and feature mappings to the other views. This file along with another feature-component mapping file, developed manually, was integrated with a component-based runtime environment based on Java OSGI dynamic technology. This runtime environment, developed by our team, allowed us to monitor the behavior of the components at run time when adding, removing or customizing tenant configurations (Figure 5).

VI. Related Work

SPLs concepts were considered, in several research, to derive customizable SaaS instance as in [7] and [8]. Analogous to their approach, ours provides a feature-oriented solution to develop a dynamically evolving SaaS platform. However, our model is developed based on service-oriented SPLs where a single SaaS application is represented as a set of services. This embodies the core concept that SaaS applications are built upon. Moreover, our platform supports multi-view evolutions where a multi-view variability meta-model is used to represent the different perspectives of the developed PIMs. This allows us to represent variability in different dimensions and provides a wider variability coverage in SaaS applications. Similarly, in [6] they proposed a conceptual model for a dynamic scalable configuration management framework based on an extended feature model (EFM), view model (VM) and configuration process model (CPM). Unlike our approach, their solution focuses only on the problem space and identifying/customizing tenants' configurations. However, in this paper we provide a model-driven evolution approach that links the problem space to the solution space via model transformations. So, we go beyond merely adding or removing features to propagating the effect of customizing a configuration to the solution space using model transformations and code generators. Moreover, they identified a limited number of models to represent variability in SaaS, whereas, in our approach we cover a wider range of variability.

Current research also proposes the use of a model-driven approach to support SPL evolution on feature-level such in [14]. Similar to our solution, their approach separates the business logic from the underlying technology, however, MDA is directly related to the architecture of the application; it, clearly, defines the parts composing the system and how they are interconnected. MDA, also, embeds the concepts of UML and other modeling standards defined by the OMG. This helps in identifying the building units of the SaaS SPL, how they interact and how they can evolve at runtime.



Your Current Configuration is:

_purchasing , _ordering , _fulfillment , _PreferredCustomer , _creditRating , _electronicCheck , _discount

Select the features you want to remove:

[✓ _PreferredCustomer] [✓ _creditRating] _electronicCheck _discount

Select the features you want to add:

_salesTax _electronicgoods _purchasing [✓ _ordering] [✓ _books] _electronics _creditCard _PreferredCustomer _regularCustomer

Fig 7. Tenant Customization

To aid SPL evolutions, authors in [15] presented re-mapping operators to overcome the negative side-effects caused by some evolutions. However, they considered the mappings between the features model, in the problem space, and the artifacts located in the solution space only while in our model, two types of mappings are maintained during evolution: (1) the mappings between tenant-specific configurations and the solution space artifacts and (2) the mappings between features in the feature model and the multi-views in the multi-view variability model.

VII. Conclusion

In this paper we have specified the different scenarios that may lead to an evolution of an SIMT SaaS instance and the tenant-level rules that govern such evolution. The presented approach provides a feature-oriented solution that lets service providers manage variability at a higher level of abstraction. The paper, also, discussed the early implementation phases of the platform exemplified by an E-commerce SaaS. The implemented feature selection module and QVT transformations embodies the infrastructure to support runtime.

As part of the ongoing research, PIM to code transformations are being implemented based on the previously mentioned phases. Then, a web-based interface will be developed and a case study will be carried out to illustrate the implemented platform. Finally, the developed platform will be tested, validated and evaluated at runtime using real-life SaaS applications.

References

[1] A. Banerjee, "A Formal Model for Multi-Tenant Software-as-a-service in Cloud Computing," in *5th ACM COMPUTE Conference: Intelligent & scalable system technologies*, 2012.

[2] J. Fiaidhi, I. Bojanova, J. Zhang and L.-J. Zhang, "Enforcing Multitenancy for Cloud Computing Environments," *IT Professional,* vol. 14, no. 1, pp. 16 - 18, 2012.

[3] M. Babar, L. Chen and F. Shull, "Managing Variability in Software Product Lines," *Software,* vol. 27, pp. 89-91, May-June 2010.

[4] D. Beuche and M. Dalgarno, "Software product line engineering with feature model.," *Methods & Tools,* pp. 9-17, 2006.

[5] "OMG Model Driven Architecture," [Online]. Available: http://www.omg.org/mda/. [Accessed 19 January 2014].

[6] J. Schroeter, P. Mucha, M. Muth, K. Jugel and M. Lochau, "Dynamic Configuration Management of Cloud-based Applications," in *SPLC '12 Proceedings of the 16th International Software Product Line Conference*, USA, 2012.

[7] S. Walraven, E. Truyen, K. Handekyn, W. Joosen and D. Landuyt, "The Efficient Customization of Multi-tenant Software-as-a-Service Applications with Service Lines," *The Journal of Systems and Software* , 2014.

[8] I. Kumara, J. Han, A. Colman, T. Nguyen and M. Kapuruge, "Realizing Service-based SaaS Applications with Runtime Sharing and Variation in Dynamic Software Product Lines," in *Services Computing (SCC), 2013 IEEE International Conference*, Santa Clara, CA, July 2013.

[9] M. Hinchey, S. Park and K. Schmid, "Building Dynamic Software Product Lines," *Computer,* vol. 45, no. 10, pp. 22 - 26, 2012.

[10] K. Kang, S. Cohen, J. Hess, W. Novak and A. Peterson, "Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report CMU/SEI-90-TR-21," Software Engineering Institute, Pittsburgh A, November 1990.

[11] G. . H. Alférez, V. Pelechano, R. Mazo, P. C. Salinesi and D. Diaz, "Dynamic adaptation of service compositions with variability modelsG.," *Systems and Software,* vol. 91, pp. 24-47, 2013.

[12] M. Abu-Matar, R. Mizouni and S. AlZahmi, "Towards Software Product Lines Based Cloud Architectures," in *The IEEE International Conference on Cloud Engineering (IC2E)*, 2013.

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

[14] A. Pleuss, B. Goetz, D. Deepak, A. Polzer and S. Kowalewskic, "Model-driven support for product line evolution on feature level," *Journal of Systems and Software,* vol. 85, no. 10, pp. 2261- 2274, October 2012.

[15] C. Seidl, F. Heidenreich and U. Aßmann, "Co-evolution of models and feature mapping in software product lines," in *SPLC '12 Proceedings of the 16th International Software Product Line Conference*, USA, September 2012.