THE NATIONAL UNIVERSITY
of SINGAPORE

School of Computing
Computing 1, 13 Computing Drive, Singapore 117417

TR20/11

Tenant Onboarding in Evolving Multi-tenant SaaS

Lei Ju, Bikram Sengupta and Abhik Roychoudhury

October 2011

T e c h n i c a l R e p o r t

Foreword

This technical report contains a research paper, development or tutorial article, which has been submitted for publication in a journal or for consideration by the commissioning organization. The report represents the ideas of its author, and should not be taken as the official views of the School or the University. Any discussion of the content of the report should be sent to the author, at the address shown on the cover.

OOI Beng Chin
Dean of School

Tenant Onboarding in Evolving Multi-tenant SaaS

Lei Ju
School of Computer Science and Technology
Shandong University, China
julei@sdu.edu.cn

Bikram Sengupta
IBM Research
India
bsengupt@in.ibm.com

Abhik Roychoudhury
National University of Singapore
Singapore
abhik@comp.nus.edu.sg

***Abstract*—A multi-tenant software as a service (SaaS) system has to meet the needs of several tenant organizations, which connect to the system to utilize its services. To leverage economies of scale through re-use, a SaaS vendor would, in general, like to drive commonality amongst the requirements across tenants. However, many tenants will also come with some custom requirements that may be a pre-requisite for them to adopt the SaaS system. These requirements then need to be addressed by evolving the SaaS system in a controlled manner, while still supporting the requirements of existing tenants. In this paper, we study the challenges associated with engineering multi-tenant SaaS systems and develop a framework to help evolve such systems in a systematic manner. We adopt an intuitive formal model of services that is easily amenable to tenant requirement analysis and provides a robust way to support multiple tenant onboarding. We perform a substantial case study of a multi-tenant blog server to demonstrate the benefits of our proposed approach.**

I. Introduction

Businesses around the world are increasingly adopting the paradigm of "Everything-as-a-Service" (XaaS). Based on the pay-per-use model of Utility Computing, XaaS frees up firms from having to commit expensive resources for computing infrastructure. Instead the resources may be acquired as and when needed as "services". These services, which may be in layers ranging from Infrastructure-as-a-Service (IaaS) at the base, to Platform-as-a-Service (PaaS), to Software-as-a-Service (SaaS), provide the foundation for *Cloud Computing* and indicate a paradigm shift in the IT world that will have far-reaching changes in how IT vendors engage with their clients going forward. The changes have both financial and technical dimensions, and will increasingly see an interplay of the two - where engineering decisions have to be founded much more strongly on economic reasoning than before. In this paper, we will study this interplay in the context of Software-as a Service or SaaS [24] - a market segment that is forecasted to reach $40.5B by 2014, when about 34% of all new business software purchases will be consumed via SaaS [2].

Informally, SaaS is described as software deployed as a hosted service by a vendor and accessed over the internet, without the need for users to deploy and maintain on-premise IT infrastructure. From a SaaS vendor's perspective, the new costs incurred through owning the SaaS infrastructure, need to be offset by leveraging the economies of scale that arise from being able to serve a high number of customers (called "tenants") from a shared, single instance of a centrally-hosted software service. Such sharing through "multi-tenancy" is feasible when the requirements of the individual tenants are similar - however, in the real world, there will always be some tenant-specific variations in requirements, and how to handle this is a key technical challenge in a multi-tenant SaaS. The usual approach has been to build in a fixed set of customization options in the software, so that each tenant may individually select an appropriate set of options at the time of on-boarding. However, this assumes that the set of all possible customizations that may be needed is known before-hand, or that the tenants will always be willing to modify their business processes to adapt to what the SaaS vendor offers, in case their desired customization is not supported. Neither of these are sound assumptions, and we expect that a multi-tenant SaaS - like all other software in the past - will need to *evolve* based on newer/differentiated capabilities demanded by large and diverse tenant organizations. Business imperatives will demand this evolution, particularly for the many medium and small-sized vendors who would be less capable to dictate the terms of engagement with their customers. Otherwise, this has the danger of slowing down the growth of SaaS, with some industry surveys having already indicated that the inability to customize SaaS applications to suit their needs is the most significant challenge that customers face with the SaaS offerings they use [14].

One may argue, of course, that tenant-specific changes go against the principle of sharing, and any such request should be handled through separate customized instances for individual tenants. Supporting such variations also increases the overhead on the SaaS vendor, and can significantly increase its costs towards maintaining the system. However, there is a large space to be covered between fully common, shared behavior to completely different, customized behavior, and we posit that in order for multi-tenant SaaS to remain viable, the real question is not whether tenant-specific changes should be supported, but to what extent

they may be accommodated within a single instance, while still retaining the benefits of sharing. This raises several interesting issues for the SaaS vendor such as how different is a new service variant being requested from the ones on offer, will it require significant new development, how will the homogeneity of the system be impacted, what are the financial trade-offs and so on. These questions call for a new way of engineering multi-tenant SaaS that allows these systems to evolve with time to cater to custom requirements from tenants, while ensuring that the commonality amongst tenants remain sufficiently high for a single shared instance to be justifiable. SaaS is an economic model for software consumption, hence this evolution will have to be grounded on the basis of financial reasoning that can benefit both the vendor - by keeping its maintenance costs within reasonable limits and increasing its profits, and the tenants - so that they can continue to benefit from lower subscription fees that result from sharing.

In this paper, we present a multi-tenant SaaS engineering approach that is motivated by the above line of reasoning. Our approach is based on an intuitive formalization of multi-tenant SaaS into a 3-level hierarchy of services, features and feature variants, and a set of tenants that wish to subscribe to existing features on offer (as shown in Fig. 1), or that place demands for new variants that need to be supported for SaaS usage. Adopting the principles of Design-by-Contract, we show how our model provides a simple yet elegant platform for defining variants of a feature, for reasoning about tenant commonality and variability, or for estimating the costs of tenant onboarding through feature/service expansion or augmentation. We then show how our model naturally supports tenant onboarding as a bi-objective optimization problem, that maximizes profit for the SaaS vendor, while striving for the best commonality or functional cohesiveness of the resulting SaaS system. Our approach is illustrated through a substantial case study of an open-source Java blog server called Apache Roller. We believe that the approach we present in this paper can provide the foundation for design and analysis toolkits that SaaS vendors may use to methodically design, refine and evolve multi-tenant SaaS systems, balancing the dual objectives of higher profits and greater commonality.

Organization: In Section II, we introduce the Apache Roller system, which is used as a running example. Section III presents a formal model for multi-tenant SaaS. We then show how to utilize the proposed model for service commonality measurement and estimating tenant onboarding costs in Section IV, which concludes with the formulation of tenant onboarding as an optimization problem with the dual objectives of increasing commonality and profit. Experiments are presented in Section V, Section VI summarizes related work, and Section VII concludes the paper.

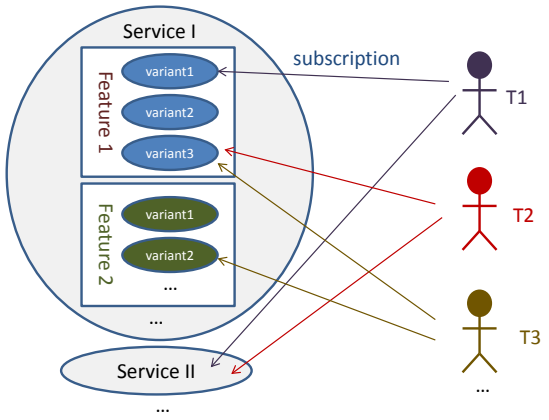

Figure 1. A SaaS system model.

II. A Working Example

In this section, we present a working example based on the Apache Roller [1] v4.0. Roller is an open source Java blog server which supports blogging features including group-blog, built-in content search engine, comments/trackbacks, RSS/Atom feeds, etc. The original Roller is a single-tenant system, where each individual weblog site needs to install, deploy, and maintain the blog server on-premises with its own infrastructure and manpower. We have rewritten (a major portion of) the Roller into a multi-tenant SaaS system to illustrate our proposed formalism. We have identified and modeled the operations/functionalities in the original Roller system as services and features in the SaaS system, as outlined below.

- Blogger service. It contains features that can be performed by an individual weblog owner, e.g., change the blog theme and template, edit or delete a blog entry, set entry permission, create an entry category, etc.
- Reader service. It contains features that can be performed by a weblog reader, e.g., search for or view a weblog entry, write comments, subscribe RSS feeds etc.
- Administration service. It is the group of administration related features e.g. user registration, login, password retrieval, user grouping/permission management, etc.

We abstract the functionality of each feature as a *contract* between the tenants and the SaaS provider, wherein the provider *ensures* some post-conditions if the tenant meets certain pre-conditions. We present below the informal descriptions of a few motivating features in an intuitive syntax popular in the Design-by-Contract (DBC) [18] community. We will formalize this in the next section.

Blogger services (S1)
Feature: editEntry (F1)/default variant(v_0)

Require
Content must be plain text or HTML format
Content must be more than 100 characters
Do
// Actual implementation goes here
Ensure
Entry is successfully stored in the data base
Notifications are sent to feeds subscribers

Blogger services (S1)
Feature: deleteEntry (F1)/default variant(v_0)

Require
Do
// Actual implementation goes here
Ensure
Entry is successfully deleted from the data base
Related comments, trackbacks, feeds are removed
Attachments are deleted from the file system

Note that a user must be the owner of the weblog/entry in order to perform any of the features in blogger service. In our model, we consider it as a service invariant to be preserved across all executions and instantiations of the service, rather than a precondition of individual features. III for the details).

Reader service (S2)
Feature: viewEntry (F2)/default variant(v_0)

Require
User must be logged in
Do
// Actual implementation goes here
Ensure
Entry and its comments are displayed
Show total number of comments (including trackbacks)

The above feature implementations constitute the default offerings (having 0 as subscript in the feature variant names) from the basic SaaS system. We will use this example implementation for explaining the different concepts in the remainder of the paper.

III. Formal Model

A. Services and Features

Formally, a SaaS component $\langle \mathcal{S}, T \rangle$ consists of a collection of *services* $\mathcal{S}$ and a set of tenants $\mathcal{T}$ that interact with the SaaS component to exercise the various SaaS features. Each service $S \in \mathcal{S}$ can be defined as a tuple $\langle F_S, \overline{\Psi_S} \rangle$, where:

- F_S is a set of features,
- $\overline{\Psi_S}$ is a set of *service invariants* over the service system state that should always be preserved.

For a service S, let the service system state be G_S. Let us assume G_S comprises of state variables $\overline{V_S}$. Clearly any service state $g \in G_S$ of S is a valuation of each variable $v \in \overline{V_S}$, that is g is a set of mappings

$$\{v \rightarrow Dom(v) \mid v \in \overline{V_S}\}$$

where $Dom(v)$ is the domain of variable v obtained from its type (we may assume only finite domain variables). Any service invariant is a quantifier free first order logic formula over $\overline{V_S}$ which should be preserved across all executions and instantiations of the service S. For example, a user must be the owner of the weblog/entries (and remain logged in) in order to use any features in the blogger service as we have discussed in Section II. Below, we formalize this notion of service invariant preservation.

Each feature F modifies or accesses its state G_S, and G_S can only be accessed via the service S. Each feature F is a tuple

$$\langle pre(F), meth(F), post(F), \overline{\Psi_F} \rangle$$

where $pre(F), post(F)$ are the pre and post-conditions of F and $meth(F)$ is the side-effect free function/method corresponding to F. $\overline{\Psi_F}$ is the set of *feature invariants* must be satisfied in all variations of feature F. We now formalize each of these components.

The pre and post conditions of a service can be defined as quantifier free first-order logic formula over the variables $\overline{V_S}$. Moreover, $meth(F)$ can be viewed as a relation $meth(F) \subseteq G_S \times G_S$, and we use the notation $g \overset{meth(F)}{\longrightarrow} g'$ to denote $(g, g') \in meth(F)$.

Any feature F (and its variations) in a service S must preserve both the service invariants $\overline{\Psi_S}$ as well as the feature invariants $\overline{\Psi_F}$. We can formalize the idea behind a service or feature invariant $\varphi \in (\overline{\Psi_S} \cup \overline{\Psi_F})$ with respect to a specific feature F as follows.

$$\forall g \in G_S \ ((g \models \varphi \wedge g \overset{meth(F)}{\longrightarrow} g') \Longrightarrow g' \models \varphi)$$

An alternative formulation can bring in concept of pre-condition and post-condition of a service by augmenting the above constraint as follows

$$\forall g \in G_S \ ((g \models \varphi \wedge g \models pre(F) \wedge g \overset{meth(F)}{\longrightarrow} g') \Longrightarrow (g' \models \varphi \wedge g' \models post(F))$$

B. Feature Variants

Usually a SaaS vendor will offer one concrete implementation to start with for each feature. However, tenants may

demand different variants of this to suit their needs, while respecting the relevant service and feature invariants. Some situations where this may arise are as follows:

1) Relaxation / Restriction of Terms of Use: A new tenant may not be agreeable to the existing pre-condition of a service feature invocation and may demand a relaxation that he can conform to. Conversely, in some cases e.g. to strengthen security related issues, the tenant may demand a stronger pre-condition.
2) Restriction / Augmentation of Feature Outcome: The tenant may demand a more restricted variant of the post-condition of the feature when it does not require its full capabilities, or may wish to augment the capabilities through a richer post-condition.

For the kinds of variation above, the SaaS vendor has two clear choices, namely (a) Create a new feature (Augmentation), or (b) Support the new variant from the existing infrastructure with required modifications (variant set expansion). We now develop a classification of the above scenarios based on which the formal definition of a feature variant will be presented. Since the functionality of a feature is encapsulated within its contract in our model, the similarity between the pre- and post- conditions of two features will determine the ease with which one may be derived from the other.

Defn 1: **[Stronger/Weaker condition:]** Let $C1$ and $C2$ be two sets of conditions (quantifier-free first-order logic formulas). We define $str(C1, C2)$ to be the set of "stronger" conditions in $C1$ w.r.t. the conditions in $C2$, i.e.,

$$str(C1, C2) = \{c \mid c \in C1 \wedge \nexists c' \in C2, c' \Rightarrow c\}$$

Similarly, set of "weaker" conditions in $C1$ w.r.t. $C2$ is

$$wkr(C1, C2) = str(C2, C1)$$

In other words, a condition clause $c \in str(C1, C2)$ is either a new requirement in $C1$ that is not captured in $C2$, or more restrictive than any condition in $C2$; while $wkr(C1, C2)$ contains the conditions in $C2$ that are either removed or relaxed in $C1$.

Defn 2: **[Distance:]** Let $s1 = \langle pre(s1), meth(s1), post(s1)\rangle$ be a new requested feature to be supported by a SaaS system, and $s2 = \langle\ pre(s2),\ meth(s2),\ post(s2)\rangle$ be an existing feature instance in the system. We define the *distance* from s_2 to s_1 as

$$D(s_1, s_2) = \alpha \cdot |wkr(post(s1), post(s2))| + \beta \cdot |str(pre(s1), pre(s2))| + \gamma \cdot |wkr(pre(s1), pre(s2))| + \delta \cdot |str(post(s1), post(s2))| \quad (1)$$

where α, β, γ, and δ are coefficients that represent the relative implementation cost for weakening a postcondition, strengthening a precondition, weakening a precondition, and strengthening a postcondition for a given SaaS feature, respectively.

We use this notion of *distance* to capture the variability between two feature requests. In particular, a smaller $D(s_1, s_2)$ usually implies less implementation effort if the new request $s1$ is to be derived from an existing implementation of s_2. It may be noted that the distance between two variants is asymmetric, i.e., $D(s1, s2) \neq D(s2, s1)$. In order to enable a fine-grained quantitative measurement, we associate a cost coefficient with each weakening/strengthening of the pre and post conditions as follows.

- **[Weaken postcondition cost α:]** In case the new request s_1 requires a weaker postcondition, the existing implementation of s_2 can still be used with relatively little modification (to hide certain functionality).
- **[Strengthen precondition cost β:]** To support a stronger precondition in the new request service s_1, the corresponding check needs to be performed before invoking an existing implementation of s_2 (i.e. $meth(s_2)$). No change is required in the method body.
- **[Weaken precondition cost γ:]** If the new request s_1 has a weaker precondition, some assumptions or assertions in the original implementation of s_2 may be violated. As a result, the developer will have to derive s_1 from implementation of s_2, by removing certain assertions/assumptions and modifying the code accordingly.
- **[Strengthen postcondition cost δ:]** If the new request s_1 requires a stronger postcondition, the developer needs to incorporate additional functionality in the implementation of $s2$ to support this.

Intuitively, weakening a postcondition or strengthening a precondition incurs less implementation cost since no modification of the existing method body is required. On the other hand, strengthening a postcondition usually requires much more implementation cost.

Defn 3: **[Feature variant:]** Let $var(F)$ contains set of all existing variants of a feature F. A new service request $s = \langle pre(s),\ meth(s), post(s), \overline{\Psi_s}\rangle$ can be considered as a variant of an existing feature variant $v = \langle pre(v), meth(v), post(v), \overline{\Psi_F}\rangle \in var(F)$ if

- v and s are bound by the *same* service and feature invariants
- $D(s, v) \leq TR$, where TR is the distance threshold defined by the SaaS vendor that manages the degree of similarity between feature variations.

If there are multiple such existing variants v in the system, we denote the one with minimal distance $D(s, v)$ as $parent(s)$. The implementation of s will be derived from $parent(s)$.

On the other hand, if a new feature request is not a variant of any existing feature instance, then the vendor will have to create a new feature in an existing service or even, in some cases, a new service to hold the feature. This is called *augmentation*, which we will discuss in Section IV. The fact that a variant of a feature satisfies the same invariants, and is within a threshold distance from the parent, helps maintain the functional cohesiveness and soundness of our model.

C. Example Feature Variants

We illustrate the feature variant formalism using the view entry feature of the reader service in our modified multi-tenant Roller system. Its default offering (variant v_0) is presented in Section II. Assume a tenant runs a weblog site where entries can be only viewed by authorized reader (as determined by the entry owner or administrator), and all comments from the entry owner should be highlighted. In our proposed model, the new request can be considered as a variant v_1 of the view entry feature, with strengthening of both the precondition and postcondition of the default variant v_0.

Reader service (S2)
Feature: viewEntry (F2)/variant(v_1)

Require
User must be logged in
User must be authorized to view this entry
Do
// Implementation can be derived from v_0
Ensure
Entry and its comments are displayed
Show total number of comments (including trackbacks)
Entry owner's comments are highlighted

The implementation of $v1$ can be derived from v_0, by adding the functionality of authorization check, as well as the capability to identify and highlight entry owner's comments. According to DEFN 2, the distance $D(v_1, v_0) = \beta + \delta$ approximately captures the implementation effort to accommodate $v1$ from v_0.

Suppose another tenant has a different perspective on the view entry feature as follows.

Reader service (S2)
Feature: viewEntry (F2)/variant(v_2)

Require
User must be logged in
Do
// Implementation can be derived from either v_0 or v_1
Ensure
Entry and its comments are displayed
Show total number of comments (including trackbacks)
Show total number of views of this entry
Entry owner's comments are highlighted

This variant v_2 can be accommodated from either the implementation of v_0 or v_1. In the first case, it requires additional code to (i)record and display the total number of entry views; and (ii) highlight entry owner's comments. As a result, the distance $D(v_2, v_0) = 2 \times \delta$. On the other hand, deriving v_2 from v_1 requires (i) simply skipping the authorization check (weakening one of the precondition); and (ii) record and display the total number of entry views (strengthening a postcondition). Thus, we have $D(v_2, v_1) = \alpha + \delta$. Intuitively, the SaaS provider may choose the second option to ease the accommodation (assuming $\alpha < \delta$). One may argue that is possible to source the different facets (pre-, post-conditions) of a request from multiple existing variants to further reduce, in principle, the implementation cost. However, we feel this will in practice, complicate development and require additional effort in integration, besides impeding a sound, systematic evolution of the SaaS system. This motivates our approach of deriving a variant from a single existing feature closest to it.

IV. How to onboard tenants?

A SaaS application has an initial set of service offerings $\mathcal{S}$, each described as a tuple $\langle F_S, \overline{\Psi_S} \rangle$ as discussed earlier. A tenant t can be characterized by a tuple $\langle requires(t), V(t) \rangle$ where $requires(t)$ is the set of service features required by tenant t and

$$V(t) : requires(t) \rightarrow \mathbb{R}^1$$

is a value estimate of the revenue that the tenant may add to the existing SaaS system by subscribing to the services it requires. In practical cases, as is intrinsic to SaaS systems, the value function will be defined on a per feature basis and in a *pay-as-you-go* manner depending on how and which features are being used by the tenant. For simplicity of the present work, we approximate the economic factor by the value estimate which is assumed to capture the projected revenue depending on the present and future usage patterns of the tenant.

Depending on the requested service features in $requires(t)$ and existing SaaS system $\mathcal{S}$, we have the following cases to accommodate each request $s \in requires(t)$:

[1] $\mathbb{R}$ is the set of real numbers

- **[Direct onboarding:]** If s requires exactly the same preconditions and postconditions as an existing variant v of some feature F in the SaaS system, and they are bounded by the same service/feature invariants, the new request s can be directly supported, without any additional change to the SaaS infrastructure.
- **[Feature expansion:]** If s is a variant of feature F in $\mathcal{S}$, s is added to F as feature expansion. The implementation of s can be derived from its closest existing variant $parent(s)$ based on the weakening/strengthen of pre and post conditions as described in Section III-B.
- **[Feature set augmentation :]** There may exist a service S in $\mathcal{S}$ such that s is bounded by the same invariants $\overline{\Psi_S}$ with S. However, s may not be a variant of any existing feature $F \in S$. In such a case, s can be added into the SaaS system as a new feature of service S with a new implementation, to onboard the tenant.
- **[Service set augmentation :]** If s is not bounded by the same invariants with any existing service in $\mathcal{S}$, then the tenant requires a new service to be defined that is then augmented with the feature s to support onboarding.

In all the cases above, the question of on-boarding depends on the SaaS vendor. We propose a decision model with two dimensions, namely the onboarding profit and level of system commonality, to help SaaS vendors decide which tenants to onboard. We introduce these dimensions next.

A. Cost Considerations

For each new tenant t to be onboarded, the net profit gained by the SaaS provider can be calculated by deducting the modification cost of the SaaS system from the revenue $V(t)$. In this section, we present a cost model to facilitate SaaS vendors estimate the cost of onboarding a new tenants. For now, we only consider the *salary* cost (on a human-hour basis) incurred by the vendor, to pay the developers who make necessary modifications to onboard a new tenant t. Intuitively, the more complex is the implementation/modification required to onboard t, the higher cost is imposed.

A new tenant t may request a set of services; this is captured in $requires(t)$. For each of the requests s and an existing SaaS system, we associate a cost to accommodate s depending on the amount of changes the existing infrastructure has to undergo to support s according to the four possible scenarios described above.

- **[Direct onboarding cost (DOC):]** An existing implementation can be directly used to support s. In this case, the estimated development cost is assumed to be a constant since this is common to all the cases below. This is the setup cost for inducting the tenant into the existing tenant base and hooking him up with his desired services and features.
- **[Feature expansion cost (FEC):]** Let $parent(s)$ be the closest existing variant of s in feature F (according to DEFN 3), we support s by feature expansion of F. Since $D(s, parent(s))$ captures the approximate implementation effort required to derive the implementation of s from $parent(s)$, the feature expansion cost FEC can be calculated as

$$FEC(s) = \rho \times D(s, parent(s)) \tag{2}$$

  where ρ is a coefficient which captures the human-hour basis salary cost per unit of change for a particular SaaS system and vendor.
- **[Feature set augmentation cost (FAC):]** This is the cost of developing and provisioning a new feature from scratch, calculated as

$$FAC(s) = \rho \times D(s, \epsilon) \tag{3}$$

  where, ϵ is an empty/null feature, and ρ is as above.
- **[Service set augmentation cost (SAC):]** This is the cost of developing and provisioning a new service (initially empty), which we assume to be a constant.

The cost of onboarding a tenant is estimated as the cumulative effort of the above costs for each new requirement, depending on the exact nature of the tenant requirements and the existing SaaS system. Multiple cost types maybe involved in a particular situation, since the tenant may demand a variant of one feature as well as a new feature of a different service. To summarize, give a new tenant $t = \langle requires(t), V(t)\rangle$, the cost of onboarding t to SaaS system $\langle \mathcal{S}, T\rangle$ can be calculated as

$$cost(t, \mathcal{S}) = \sum_{\forall s \in requires(t)} DOC|FEC(s)|FAC(s)|SAC \tag{4}$$

where DOC and SAC are the constant cost of direct onboarding and service set augmentation, respectively; FEC is the cost that implements s from its nearest variant in the existing SaaS system, while FAC is the implementation cost when s is implemented directly.

Example 1: Given an example SaaS service

$$S = \langle F1 : \{v_0, v_1\}, F2 : \{v_0\}, F3 : \{v_0, v_1\}\rangle$$

Assume the system has two onboarded tenants $T1$ and $T2$, and the feature (variant) subscription of $T1$ and $T2$ are $\langle F1 : \{v_0\}, F2 : \{v_0\}, F3 : \{v_0\}\rangle$ and $\langle F1 : \{v_1\}, F3 : \{v_1\}\rangle$, respectively. Let us consider the onboarding cost for the following 3 tenants as below:

- *requires(T3) = $\langle F1 : \{v_0\}, F2 : \{v_0\}, F3 : \{v_1\}\rangle$. The onboarding cost for T1 is $3 \times DOC$.*

- *requires(T4) =* $\langle F1 : \{v_2\}, F3 : \{v_0\}\rangle$*. Assume that the closest existing variant of* v_2 *is* v_1 *in* $F1$ *(*$parent(v_2) = v_1$*), we have the onboarding cost for T2 to be* $\rho \times D(v_2, v_1) + DOC$.
- *requires(T5) =* $\langle F1 : \{v_1\}, F4 : \{v_0\}\rangle$*. The onboarding cost for T3 is* $DOC + D(v_0, \epsilon)$.

According to our model, for the given 3 tenant, onboarding tenant $T5$ *incurs the highest cost.*

B. Commonality Considerations

An important factor that characterizes the level of reuse of a SaaS system is the notion of *commonality* of a service. A SaaS vendor may be offering several features within a service, with multiple variants in each feature. To offset development costs, the SaaS vendor would want each feature variant to be subscribed to by as many tenants as possible. Commonality of a service measures the fraction of tenants that subscribe to a feature variant on average.

Let N^f denote the total number of tenant subscriptions to feature f (across all its variants). Let $|var(f)|$ denote the total number of variants of feature f. We now define commonality as follows.

Defn 4: **[Commonality (original):]** *The commonality* $Comm_S$ *of a SaaS service* S *consisting of a set of features* F_S *and a set of subscribing tenants* T_S *(*> 0*) is given by:*

$$COMM_S = \frac{\sum_{f \in F_S} N^f}{|T_S| * \sum_{f \in F_S} |var(f)|}$$

Intuitively, the more the number of existing variants a tenant subscribes to, the higher the commonality the SaaS system will have. In general, as new tenants onboard, the SaaS vendor would like to ensure that the commonality of a service remains above a threshold, which we call the **commonality threshold**.

Example 2: Consider the example SaaS system discussed in Example 1. The initial system (with tenants $T1$ *and* $T2$ *onboarded) has total 5 subscriptions, 2 tenants, and 5 feature variants. Thus, the commonality of the initial system is* $\frac{5}{2*5} = 0.5$*. If tenant* $T3$ *is then onboarded, the system will have total 8 subscriptions and 3 onboarded tenants, while the number of variants remains to be 5. Hence, the new commonality becomes* $\frac{8}{3*5} = 0.533$*. Similarly, the commonality after onboarding tenants* $T4$ *and* $T5$ *will be* $\frac{10}{4*6} = 0.417$ *and* $\frac{12}{5*7} = 0.343$*, respectively.*

C. Profit and Commonality Considerations

When a new tenant wants to onboard an existing SaaS system, both profit and commonality need to be considered. The net profit is the net value obtained by onboarding a tenant t, calculated by deducting the development cost of onboarding t from $V(t)$. Even if commonality does not improve (e.g. a new tenant may need a small subset of the services/features), the tenant may still be good to onboard if there is no major development cost, considering the increment in net profit as long as the new commonality remains above the threshold.

Defn 5: **[Onboarding a single tenant:]** *Given a tenant* $t = \langle requires(t), V(t)\rangle$ *and a commonality threshold* $COMM$ *on an existing SaaS system* $\langle \mathcal{S}, T\rangle$*,* t *can be onboarded if*

- $V(t) - cost(t) \geq 0$*, and*
- *Each service in the resulted new SaaS system remains above* $COMM$.

D. Generic Tenant Onboarding Problem

In Section IV-C we consider the onboarding activity for a standalone tenant. In the general case, there might be multiple tenants waiting to be onboarded. The SaaS vendor will periodically review all tenant requests, and come up with a development and onboarding plan. Having a group of tenants to select from gives the vendor greater opportunities to identify commonalities in the requests and optimize development - a feature request that may not appear to offer good financial returns when viewed from the perspective of a single tenant, may seem like a sound investment when several tenants are looking to subscribe to it (or variants close to it). At the same time, onboarding tenants purely from the profit perspective without regards to commonality may lead to excessive variants that make the system difficult to maintain. This duality gives rise to an optimization problem for onboarding tenants.

Given a SaaS system $\langle \mathcal{S}, T\rangle$ and a new set of tenants $\hat{T}$ waiting to be onboarded, let $\Gamma(\hat{T})$ denote the net profit obtained by onboarding $\hat{T}$, resulting in a new system $\langle \mathcal{S}', T \bigcup \hat{T}\rangle$, and let C be a vector denoting the commonality of the services in $\mathcal{S}'$. $\Gamma(\hat{T})$ is calculated as follows:

$$\sum_{t \in \hat{T}} V(t) - cost(\hat{T}, \mathcal{S}') \tag{5}$$

where $cost(\hat{T}, \mathcal{S})$ lifts the definition in Equation 4 to a set of tenants, and may be computed after taking into account similarities in the requests of new tenants, such that development costs for new features and variants can be optimized. For example, if more than one new tenant require a particular service feature that cannot be derived from the existing system, we consider FAC for one of the tenants, and only DOC for the rest. Similarly, if a feature

request from a new tenant can be derived more easily out of a request from another new tenant than from the existing system, FEC is reduced.

Defn 6: **[Profit-Commonality Maximizing tenant subset:]** *Given a set of potential tenants $T_p = t_1 \dots t_k$, each characterized as a tuple $\langle requires(t_i), V(t_i)\rangle$ and an existing SaaS system $\langle \mathcal{S}, T\rangle$, the onboarding problem involves selecting the tenant* subset $\hat{T} \subseteq T_p$ *that maximizes profit $\Gamma(\hat{T})$ and leads to the best commonality of the resulting SaaS system.*

The net profit and commonality can be calculated as previously for each possible subset $\hat{T}$. The multi-objective nature of the above optimization problem suggests that a solution candidate is described by a vector quantity. If a vector quantity for solution candidates is used, improvement in these solutions should occur only when some objective (either the net profit or any element in the commonality vector C) improves without degradation in the remaining objectives. If this is not possible, then the current solution is said to be optimal in the Pareto optimal sense or nondominated. The set of all Pareto optimal solutions is known as the Pareto optimal set or Pareto front. Different techniques have been proposed for multi-objective Pareto front computation [12], [26].

V. Case Study

In this section, we present a case study of the multi-tenant Roller system as described in Section II. The Roller SaaS system consists of three top-level services Blogger Service (S1), Reader Service (S2), and Administration Service (S3). In each service, the supported operations/functionalities are modeled as features according to our proposed formalism. For each feature, we list a predetermined set of all possible pre and post conditions that can be chosen for it. This lets us manage the tenant requirements elicitation process. A tenant who wants to subscribe to a feature can select a subset from the listed pre and post conditions. As a result, by choosing different combinations among them, we are able to build a diversity of variants of each feature. The total number of possible variants can be different for various features in our Roller SaaS system. For example, the viewEntry feature in reader service has 12 possible variants to satisfy the diversified needs of blog entry display. On the other hand, the login feature in administration service has only 2 possible variants (i.e., whether or not to remember the login info) in our SaaS system.

A. Commonality of a Service

We first discuss the change in commonality as the system evolves with more tenants getting onboarded. Figure 2 shows

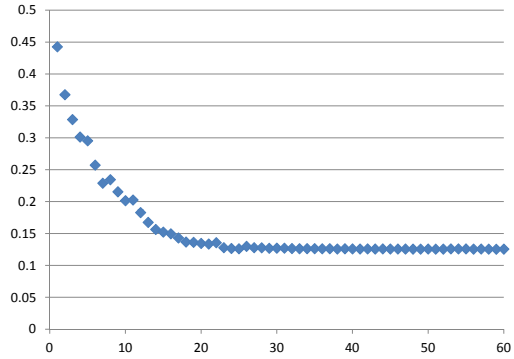

Figure 2. Change of commonality while tenants onboard.

the results obtained for the Reader Service of our Roller SaaS system. The service initially contains 9 distinct features (with one default variant for each feature). According to our predefined list of pre and post conditions for each feature, we have total 56 possible variants defined for these features. We assume that each tenant t randomly subscribes to at least 5 features in the service, and requests a single variant of each feature it subscribes to. For each required feature $F \in requires(t)$, we randomly select one or more pre and post conditions from the predefined condition list of the feature. The onboarding process in handled as per the approach described in (Section IV). The results shown in Figure 2 is obtained *without* any tenant selection mechanism, i.e., every randomly generated tenant is accommodated.

When the first few tenants are onboarded to the system, it is unlikely that a new required feature variant already exists in the system. New variants, features, or even services need to be supported to expand the system. Hence, we observe a sharp decline in the service commonality. It is a reasonable price that a SaaS provider has to pay in the business startup phase. While more and more tenants get onboarded, the SaaS system matures such that most of the new requirements can be supported with existing feature variants. As a result, commonality of the service becomes steady and approximately converges around $\frac{7}{56} = 0.125$, where 7 is the average subscriptions per tenants and 56 is the total number of possible feature variants in the given service.

B. Tenant Selection

In reality, a SaaS vendor may want to maximize profit while maintaining a reasonable degree of commonality. Hence we evaluate selective tenant onboarding using the bi-objective criteria explained in Section IV, on the Reader Service of the Roller SaaS system. We consider a scenario where 20 randomly generated potential tenants ($|\hat{T}| = 20$) request onboarding to a system with 10 existing tenants.

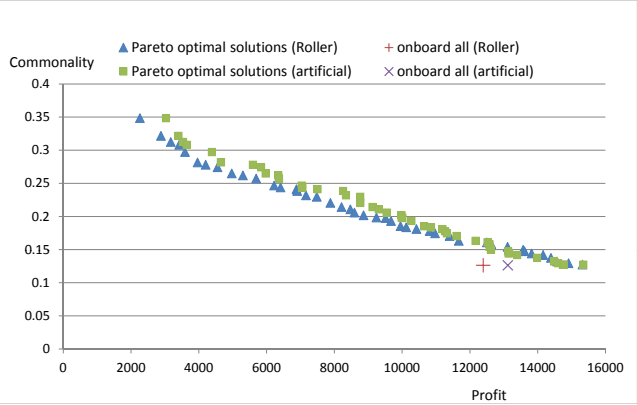

Figure 3. Pareto optimal solution for profit/commonality maximization.

Similarly as in Section V-A, assume each tenant randomly subscribes some of the 9 features in the reader service, where each subscription is associated to one of the predefined 56 possible variants in our system. As discussed in Section IV, the problem is to identify a subset of tenants $\hat{T}' \subseteq \hat{T}$, such that the resulted new system is Pareto optimal w.r.t the profit and commonality.

In this experiment, we adopt a subscription-based pricing model. We assume the estimated revenue of onboarding a tenant t is proportional to the number of features t subscripts, i.e.,

$$V(t) = RPS * |requires(t)|$$

where RPS is a constant factor indicates the revenue for each feature subscription, which is assumed to be 150 in this experiment. in Section IV-A. We consider only the implementation cost required to support a new tenant. However, other cost structure (e.g., hardware, maintenance, etc.) can be easily adopted in our model. In our experiments, we set the values for each of the following cost parameters based on our hands-on experience when converting the Roller into a multi-tenant SaaS system. In particular, the cost is computed on account of the man-hour to complete a corresponding task and the wage of a system engineer.

- DOC, the direct onboarding cost, is set to 40.
- We set the coefficients α, β, γ, and δ in DEFN 2 to be 3, 5, 8, and 10, respectively. We use these coefficients to show approximately the relative implementation cost of weakening/strengthening a pre/post condition in the variant. The relative values are obtained by measuring and comparing the *average* efforts for each of the four activities during the SaaS system conversion, in terms of locating the place to make changes, number of files and line of code to edit, as well as testing to ensure the correctness of the modification. Finally, the coefficient ρ in Equation 2 is set to be 20.
- The distance threshold in DEFN 3 is set to 40. We do not consider the SAC cost in this experiment.

Note that the above setting of parameter values is based on assumption and our experience in converting the Roller into multi-tenant SaaS system. It may not be generally applicable. However, our proposed tenant onboarding model makes *no* assumption about the relative or absolute values of the parameters. Values can be assigned to the parameters according to the real situation. Furthermore, we will show in Section V-C that our tenant selection is not particularly sensitive to the absolute values of these parameters.

For each possible subset $\hat{T}' \subseteq \hat{T}$, the profit of onboarding $\hat{T}'$ can be calculated with Equation 5 in Section IV-D. The commonality of resulted new system can be calculated as in Definition 4. We maintain a set of Pareto optimal solutions, such that there is no other subsets of $\hat{T}$, if onboarded, will result in *both* higher profit and higher commonality. In our current implementation, we exhaustively find each possible subset $\hat{T}' \subseteq \hat{T}$, and compute the profit and commonality if the subset is onboarded to the initial system. The total analysis time for the Pareto optimal computation when $|\hat{T}| = 20$ takes less than 1 minute on a Intel(R) Xeon(TM) 2.20 Ghz processor with 2.5 GB of RAM. However, if a larger group of onboarding tenants has to be considered, evolutionary algorithms ([12]) can be implemented for a more efficient Pareto front computation.

Figure 3 shows our experimental results for the above-mentioned Roller system setup. The ▲ marks (labeled with "Pareto optimal solutions (Roller)") represent the Pareto optimal solutions. Each Pareto optimal solution is associated with the corresponding subset of tenants to be onboard out of the 20 tenants in $\hat{T}$. Such a Pareto front representation allows SaaS provider to onboard potential tenants selectively, which results in good profit while retaining enough commonalities to exploit the economies of scale attained. Figure 3 also shows a dominated solution corresponding to onboard all the 20 tenants in $\hat{T}$, with $Profit = 12390$ and $commonality = 0.1264$ (the $+$ mark labeled with "onboard all (Roller)"). The SaaS vendor may avoid this solution due to the poor commonality in the resultant system. In this experiment, we use only one single service (the reader services in the COMS) to illustrate our framework. However, for the entire SaaS system with multiple services, the optimization problem can be easily adopted at the system level by considering the average commonality of all services.

C. Sensitivity of the Proposed Mechanism

In Section V-B we have evaluated our model using the Roller SaaS system. The values of the parameters are set based on our relevant experience during conversion of the system. In this section, we would like to analyze the sensi-

tivity of our tenant selection mechanism w.r.t. the absolute values of these parameters.

We have performed the tenant selection for the same initial system and set of tenants to be onboarded, with the following completely different setting of the parameters:

$$DOC = 20, \alpha = 2, \beta = 6, \gamma = 6, \delta = 7, \rho = 20$$

The resultant Pareto optimal solutions (the ■ marks labeled with "Pareto optimal solutions (artificial)") and the result of onboarding all 20 tenants (the $\times$ marks labeled with "onboard all (artificial)") are also shown in Figure 3.

We observe quite similar trends in the two sets of computed solutions. It shows that given the same initial system and tenants to be onboarded, the tenant selection problem is not very sensitive to the absolute values of the parameters in our model (as long as the relative relations between them remain the same). In other words, our experimental results show that if a subset of tenants is considered to be a good candidate to onboard to a given system, it is very likely that onboarding the same subset of tenants under a different cost scheme also produces high profit and good commonality.

VI. Related Work

Many existing work on multi-tenant applications focus on shared data architecture and security management [10], [13], [3], [25]. There has been relatively little research so far on the impact tenant variability may have on the functionality and evolution of a SaaS system over its lifecycle. This is not surprising given that SaaS is a relatively recent phenomenon, and hence the initial focus is bound to be on issues that are related directly to its feasibility (such as security or performance). However, the fact that a SaaS system needs to functionally cater to multiple tenants is now increasingly understood, leading to research on how to model variability in a SaaS, and how to make a SaaS system more customizable. Models and techniques successfully employed in software product line engineering have been applied in multi-tenant systems to manage configuration and customization of service variants [11], [19], [20]). In particular, [20] extends the variability modeling ([4]), which provides information for a tenant to choose/customize the SaaS application and guides the SaaS provider for service deployment. Maturity models for SaaS systems have been studied in [16].

In the context of software product line engineering ([22]), feature models are widely used to capture the feature information of individual products, and possible relationship (parent/child, inclusive/exclusive) between them. Various automated analysis methods have been proposed to analyze the characteristic of feature models (see [5] for a survey). Automated tools have also been developed to (i) identity and propagate changes between locations of different versions of a product line architecture ([9]); and (ii) generate test cases for software product line validation ([21]).

The work of [6] discusses potential challenges in implementation / maintenance of multi-tenant systems. It presents an architectural approach which tries to separate the multi-tenant configuration and underlying implementation as much as possible, by adopting the 3-tiers architecture (authentication, configuration, and database) in the traditional single-tenant web application. Along the same lines, experiences in modifying industrial-scale single-tenant software systems to multi-tenant software have been reported in [7], [8].

The work of [15] develops a multi-tenant placement model which decides the best server where a new tenant should be accommodated. In principle, a new tenant will be placed on the server with minimum remaining residual hardware resource left that meets the resource requirement of the new tenant. Finally, we have primarily studied functionality issues for managing multi-tenant SaaS in this paper. There have also been studies on service performance issues in multi-tenant SaaS (*e.g.*, see [17]).

Our previous work [23] serves as a position paper on multi-tenant SaaS. In [23], we had mentioned possible research directions for multi-tenant SaaS, such as a model for multi-tenant SaaS to evolve in a controlled manner, and issues in testing multi-tenant SaaS. In this paper, we have addressed the first set of issues, namely building a model of multi-tenant SaaS to control its evolution.

VII. Conclusion

In this paper, we have presented a formal framework for managing the evolution of multi-tenant SaaS systems. Our approach is based on reasoning about tenant requirement variability and commonality in terms of their contracts (pre- and post-conditions). We have discussed different scenarios that arise when accommodating a new tenant in a SaaS, and the associated development costs. The tenant onboarding problem has been motivated as a bi-objective optimization problem wherein the SaaS vendor tries to maximize profit while retaining a satisfactory degree of commonality. In future, we plan to study other aspects of multi-tenant SaaS evolution, including testing and re-factoring such systems, as outlined in our position paper [23].

Acknowledgements

This work was partially supported by an IBM Faculty Award, and a Singapore Ministry of Education research grant MOE2010-T2-2-073. Seth Hetu helped the first author in understanding the Apache Roller case study.

REFERENCES

[1] Apache roller. http://roller.apache.org.

[2] Worldwide Software as a Service 2010-2014 Forecast: Software Will Never Be The Same. *IDC Report*, Doc223628, 2010.

[3] S. Aulbach, T. Grust, D. Jacobs, A. Kemper, and J. Rittinger. Multi-tenant databases for software as a service: schema-mapping techniques. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pages 1195–1206, 2008.

[4] J. Bayer and et al. Consolidated product line variability modeling. *Software Product Lines*, pages 195–241, 2006.

[5] D. Benavides, S. Segura, and A. Ruiz-Cortés. Automated analysis of feature models 20 years later: a literature review. *Information Systems*, 35(6):615–636, 2010.

[6] C. Bezemer and A. Zaidman. Multi-Tenant SaaS Applications: Maintenance Dream or Nightmare? In *Proceedings of the 4th International Joint ERCIM/IWPSE Symposium on Software Evolution (IWPSE-EVOL)*, 2010.

[7] C. Bezemer, A. Zaidman, B. Platzbeecker, T. Hurkmans, and A. Hart. Enabling multi-tenancy: An industrial experience report. In *Intl. Conf. on Software Maintenance (ICSM)*, 2010.

[8] C.-P. Bezemer and A. Zaidman. Challenges of reengineering into multi-tenant SaaS applications. Technical Report TUD-SERG-2010-012, Delft University of Technology, 2010.

[9] P. Chen, M. Critchlow, A. Garg, C. Van der Westhuizen, and A. van der Hoek. Differencing and merging within an evolving product line architecture. *Software Product-Family Engineering*, pages 269–281, 2004.

[10] F. Chong, G. Carraro, and R. Wolter. Multi-Tenant Data Architecture. *MSDN Library, Microsoft Corporation*, 2006.

[11] K. Czarnecki, M. Antkiewicz, and C. Kim. Multi-level customization in application engineering. *Communications of the ACM*, 49(12):65, 2006.

[12] K. Deb. *Multi-objective optimization using evolutionary algorithms*. John Wiley & Sons, 2001.

[13] C. Guo et al. A framework for native multi-tenancy application development and management. *9th IEEE Intl. Conf. on E-Commerce Technology and 4th IEEE Intl. Conf. on Enterprise Computing, E-Commerce and E-Services (CEC-EEE)*, 2007.

[14] J. M. Kaplan. How SaaS is Overcoming Common Customer Objections. *Cutter Consortium: Sourcing and Vendor Relationships. Executive Update*, 8(9), 2008.

[15] T. Kwok and A. Mohindra. Resource Calculations with Constraints, and Placement of Tenants and Instances for Multi-tenant SaaS Applications. *Intl. Conf. on Service Oriented Computing (ICSOC)*, 2008.

[16] T. Kwok, T. Nguyen, and L. Lam. A software as a service with multi-tenancy support for an electronic contract management application. In *IEEE SCC*, 2008.

[17] X. Li, T. Liu, Y. Li, and Y. Chen. SPIN: Service performance isolation infrastructure in multi-tenancy environment. *International Conference on Service-Oriented Computing (ICSOC)*, pages 649–663, 2008.

[18] B. Meyer. Applying "design by contract". *Computer*, 25(10):40–51, 1992.

[19] R. Mietzner and F. Leymann. Generation of BPEL customization processes for SaaS applications from variability descriptors. In *Proceedings of the IEEE International Conference on Services Computing*, volume 2, pages 359–366. IEEE Computer Society, 2008.

[20] 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 *Proceedings of the ICSE Workshop on Principles of Engineering Service Oriented Systems*, 2009.

[21] G. Perrouin, S. Sen, J. Klein, B. Baudry, and Y. Le Traon. Automated and scalable T-wise test case generation strategies for software product lines. *International Conference on Software Testing, Verification and Validation*, pages 459–468, 2010.

[22] K. Pohl, G. Böckle, and F. Van Der Linden. *Software product line engineering: foundations, principles, and techniques*. Springer-Verlag New York Inc, 2005.

[23] B. Sengupta and A. Roychoudhury. Engineering multi-tenant software-as-a-service systems. In *ICSE Workshop on Principles of Engineering Service Oriented Systems (PESOS)*, 2011.

[24] M. Turner, D. Budgen, and P. Brereton. Turning software into a service. *Computer*, 36(10):38–44, 2003.

[25] C. Weissman and S. Bobrowski. The design of the force.com multitenant internet application development platform. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pages 889–896, 2009.

[26] E. Zitzler and L. Thiele. Multiobjective evolutionary algorithms: A comparative case study and the strength pareto approach. *IEEE transactions on Evolutionary Computation*, 3(4):257, 1999.