Tenant Onboarding in Evolving Multi-tenant Software-as-a-Service Systems

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
School of Computing
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 focus on functional variability amongst tenants in a multi-tenant SaaS and develop a framework to help evolve such systems systematically. 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, which is modeled as a bi-objective optimization problem that attempts to maximize vendor profit and tenant functional commonality. 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 study this interplay in the context of Software-as a Service or SaaS [19].

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 is overly restrictive and the inability to customize SaaS applications to suit their needs is the most significant challenge that customers face with the SaaS offerings they use [21].

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 Figure 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 (almost 90,000 lines of Java code). This system was originally designed as a single-tenant system for on-premise usage, but we extended it to support multiple tenants as per our model. 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.

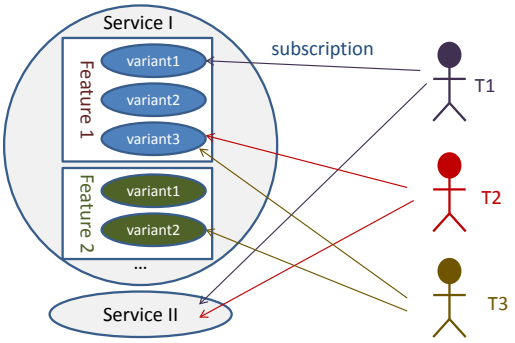

Figure 1. A SaaS system model.

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 rich blogging features. The original Roller is a single-tenant system, and 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 part 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 performed by an individual weblog owner, e.g., change the blog theme, edit/delete a blog entry, create entry category etc.
- **Reader service**. It contains features performed by the weblog readers, e.g., search for or view a weblog entry, write comments, subscribe to RSS feeds etc.
- **Administration service**. It is a group of administration related features e.g. user registration, login, password retrieval, 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 description of a few motivating features in an intuitive syntax popular in the Design-by-Contract (DBC) [14] 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
Ensure
Entry is successfully stored in the data base
Notifications are sent to feeds subscribers

Blogger services (S1)
Feature: deleteEntry (F2)/default variant(v_0)

Require
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.

Reader service (S2)
Feature: viewEntry (F3)/default variant(v_0)

Require
User must be logged in
Ensure
Entry and its comments are displayed
Show total number of comments

The above example feature contracts (with associated implementations not shown) constitute the default offerings (having 0 as subscript in the feature variant names) from the basic SaaS system. We will use this example to illustrate 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.

Each feature $F \in F_S$ is a tuple $\langle pre(F), meth(F), post(F) \rangle$, where $pre(F), post(F)$ are the pre and post-conditions of F, and $meth(F)$ is the method implementation corresponding to F.

B. Feature Variants

Usually a SaaS vendor will offer one concrete implementation to start with for each feature (e.g., the default variants listed in Section II). 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 less 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, the set of "weaker" conditions 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 the constant cost coefficients for a given SaaS feature.

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 features 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 service request 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. 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 $s1 = \langle pre(s1),\ meth(s1), post(s1)\rangle$ can be considered as a variant of an existing feature variant $s2 = \langle pre(s2), meth(s2), post(s2)\rangle \in var(F)$ if

- $s1$ and $s2$ are bound by the *same* service invariants, and
- $D(s1, s2) = \min\{D(s1, v) | \forall v \in var(F) \wedge\ D(s1, v) \leq TR\}$, where TR is the distance threshold defined by the SaaS vendor that manages the degree of similarity between feature variations.

We denote $s2 = parent(s1)$, such that $s2$ is closest to $s1$ among all existing feature variants. The fact that a variant of a feature satisfies the same invariants, and is within a threshold distance from its parent, helps maintain the functional cohesiveness and soundness of our model. Otherwise, if no such $s2$ exists in the system, the vendor will have to create a new feature/service to hold the the new request. This is called augmentation, which we will discuss in Section IV.

C. Example Feature Variants

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 model, this new request can be considered as a variant of the default offering $S2/F3/v0$ as presented in Section II, with strengthening of both the precondition and postcondition of v_0.

Reader service (S2)
Feature: viewEntry (F3)/variant(v_1)

Require
User must be logged in
User must be authorized to view this entry
Ensure
Entry and its comments are displayed
Show total number of comments
Entry owner's comments are highlighted

The implementation of $v1$ ($meth(v1)$) can be derived from $meth(v_0)$, by adding the functionality of authorization, 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 (F3)/variant(v_2)

Require
User must be logged in
Ensure
Entry and its comments are displayed
Show total number of comments
Show total number of views of this entry
Entry owner's comments are highlighted

This variant v_2 can be accommodated from either the existing 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. Hence, $D(v_2, v_0) = 2 \times \delta$. 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. Onboarding 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 $s \in requires(t)$:

- **[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 augmented.

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

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

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 tenant. For now, we only consider the *implementation* cost 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 is the cost imposed.

For each of the new requests $s \in requires(t)$ 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 small constant setup cost.
- **[Feature expansion cost (FEC):]** In this case, s can be derived from the implementation of $parent(s)$ (according to DEFN 3). The development cost of s is proportional to the distance between s and $parent(s)$:

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

  where ρ is a coefficient which captures the human-hour basis salary cost per unit of change for the 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. To summarize, given 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

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

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:]** *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** (denoted as $COMM$). This captures the overall degree of functional commonality that the system vendor desires to maintain across tenants.

C. Profit and Commonality Considerations

Based on our cost model and the commonality measure, the SaaS provider may choose to onboard a tenant t if (i) there is increment in the net profit (i.e., $V(t) - cost(t, \mathcal{S}) \geq 0$); and/or (ii) the resulting system has a new commonality value that remains above the commonality threshold ($COMM$) determined by the SaaS vendor.

In a general case, there might be multiple tenants waiting to onboard. The SaaS vendor will periodically review all tenant requests, and devise 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 new 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 5: **[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 commonality) 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.

V. Case Study

In this section, we present a set of experiments on the Apache Roller system as described in Section II. The original Roller weblog consists of 569 Java/xml source files and *close to 90000 lines of code (LOC)*. In our case study, we have modified *around 180 sources files* to evaluate our proposed multi-tenant SaaS management approach. In particular, we use the Reader Service to evaluate our tenant selection model. The service initially contains 9 distinct features (with one default variant for each feature). For each feature we identified, we list a predetermined set of all possible pre and post conditions that can be chosen for it. 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. According to our predefined list of pre and post conditions for each feature, we have total 56 possible variants defined for these features in Reader Service. 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.

A. Tenant Selection

We show how our proposed approach can provide practical guidance to the SaaS designer in an onboarding decision. We consider a scenario where 20 randomly generated potential tenants ($|\hat{T}| = 20$) request onboarding to a system with 10 existing tenants. As discussed in Section IV, the problem is to identify a subset of tenants $\hat{T}' \subseteq \hat{T}$, such that the resulting 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 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, we measure the average LOC to modify for each activity, as well as the difficulty to locate and make the modification (e.g., the Cyclomatic Complexity and file modified). 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. For example, the hardware/infrastructure costs should be added in Equation 5, which are related to the hardware capacity expansion for a group of onboarding tenants. In our experiments,

- 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.
- 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.

For each possible subset $\hat{T}' \subseteq \hat{T}$, the profit of onboarding $\hat{T}'$ can be calculated with Equation 5 in Section IV-C. The commonality of the resulting new system can be calculated as in Definition 4. We maintain a set of Pareto optimal solutions, such that there is no other subset of $\hat{T}$, which, if onboarded, will result in *both* higher profit and higher

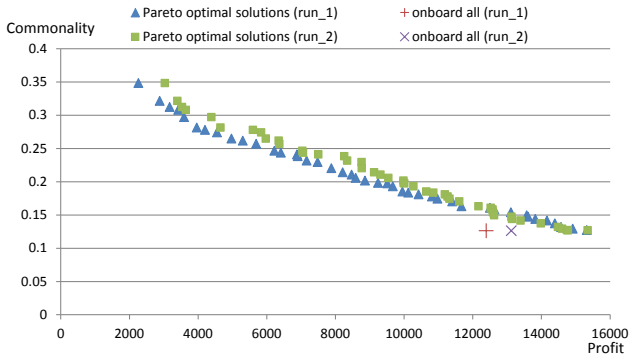

Figure 2. Pareto optimal solution for profit/commonality maximization.

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 can be implemented for a more efficient Pareto front computation.

Figure 2 shows our experimental results for the above-mentioned Roller system setup. The ▲ marks (labeled with "Pareto optimal solutions (run_1)") represent the Pareto optimal solutions. Each Pareto optimal solution is associated with a subset of tenants to be onboarded out of the 20 tenants in $\hat{T}$. Such a Pareto front representation allows SaaS provider to carefully explore the trade-offs between commonality and profit when given a large set of tenants to onboard. If the provider decides on a commonality threshold that must be maintained, then the Pareto front immediately suggests the subset of tenants whose onboarding can maximize profitability while preserving sufficient commonality. Figure 2 also shows a dominated solution corresponding to onboarding of all 20 tenants in $\hat{T}$, with $Profit = 12390$ and $commonality = 0.1264$ (the $+$ mark labeled with "onboard all (run_1)"). 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 Service) 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.

B. Sensitivity of the Proposed Mechanism

In Section V-A, the setting of parameter values is based on our experience on a particular 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. In this section, we would like to analyze the sensitivity 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 (run_2)") and the result of onboarding all 20 tenants (the $\times$ marks labeled with "onboard all (run_2)") are also shown in Figure 2. We observe quite similar trends in the two sets of computed solutions. It shows that our approach is robust since 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 parameter relations remain, similar results can be obtained. The more accurate the parameters, however, better can the cost/profit be estimated.

VI. Related Work

Many existing work on multi-tenant applications focus on issues related to their basic feasibility such as shared data architecture and security management [6], [8], [20]. From the software engineering perspective, 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 ([7], [15], [16]). In particular, [16] extends variability modeling ([2]) to provide information for a tenant to choose/customize the SaaS application and guides the SaaS provider during service deployment. Feature diagrams (introduced in [9]) are often used to model and analyze software product lines, and allows various types of relationships (parent/child, mandatory/optional/alternative etc.) between features and feature trees to be represented. Such an approach can be incorporated within our model if we wish to impose constraints on sets of features that may or may not occur together. Feature models have also been used to support variability modeling and quality-of-service analysis of web service orchestrations [10]. It may be noted, however, that unlike existing work, we use features to reason about commonality and variability in support of SaaS *evolution* and tenant onboarding, rather than to offer a fixed set of customization options to any tenant.

[3] presents an architectural approach which tries to separate multi-tenant configuration and underlying implementation, by adopting a 3-tier architecture (authentication, configuration, 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 [4]. A recent work [17] focuses on the design and implementation of a multi-tenant workflow engine that enables multiple tenants to run

their workflows securely within the same engine instance. An earlier work [12] presented a case study of a multi-tenant electronic contract management SaaS. The multi-tenant placement problem has been discussed in [11] and [22], which decides the best server where a new onboarded tenant should be accommodated. In contrast, we are interested in identifying the subset of tenants that should be onboarded and understanding how the SaaS system needs to evolve to accommodate this. 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 [13]). The cost assignment in our model can be improved, currently it is related to the salary of the developer. One can investigate more complex economic value assignment, such as those prescribed by service value networks [5].

In our earlier work [18], 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. No technical solutions were presented in [18]. In this paper, we have provided technical solutions for the first set of problems, 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. A detailed case study based on the Apache roller system demonstrates the usefulness of our approach.

Acknowledgements

This work was partially supported by an IBM Faculty Award, and a FRC grant T1 251RES0914 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] J. Bayer and et al. Consolidated product line variability modeling. *Software Product Lines*, pages 195–241, 2006.

[3] 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.

[4] 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.

[5] N.S. Caswell et al. Estimating value in service systems: A case study of a repair service system. *IBM Systems Journal*, 47(1), 87-100, 2008.

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

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

[8] C. Guo et al. A framework for native multi-tenancy application development and management. *9th IEEE CEC-EEE*, 2007.

[9] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical report, Software Engineering Institute, Carnegie Mellon University, 1900.

[10] A. Kattepur, S. Sen, B. Baudry, A. Benveniste, and C. Jard. Variability modeling and QoS analysis of web services orchestrations. In *IEEE International Conference on Web Services (ICWS)*, pages 99–106. IEEE, 2010.

[11] 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.

[12] 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.

[13] 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.

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

[15] 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.

[16] 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.

[17] M. Pathirage, S. Perera, I. Kumara, and S. Weerawarana. A multi-tenant architecture for business process executions. In *IEEE International Conference on Web Services (ICWS)*, pages 121–128. IEEE, 2011.

[18] 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.

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

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

[21] J. Kaplan. How SaaS is overcoming common customer objections. *Cutter Consortium: Sourcing and Vendor Relationships.*, 8(9), 2008.

[22] Y. Zhang, Z. Wang, B. Gao, C. Guo, W. Sun, and X. Li. An effective heuristic for on-line tenant placement problem in SaaS. In *IEEE International Conference on Web Services (ICWS)*, pages 425–432. IEEE, 2010.