THE NATIONAL UNIVERSITY
of SINGAPORE

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

TR10/10

Managing Tenants in a Multi-tenant SaaS

Lei Ju, Ansuman Banerjee, Abhik Roychoudhury and Bikram Sengupta

October 2010

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

Managing Tenants in a Multi-tenant SaaS

Lei Ju[1] Ansuman Banerjee[2] Abhik Roychoudhury[1] Bikram Sengupta[3]
[1]National University of Singapore(NUS) [2]Indian Statistical Institute [3]IBM Research, India
{julei,abhik}@comp.nus.edu.sg, ansuman@isical.ac.in, bsengupt@in.ibm.com

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 reuse, 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 prerequisite 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 and validate such systems in a systematic manner. We adopt an intuitive formal model of services. We show that the proposed formalism is easily amenable to tenant requirement analysis and provides a systematic way to support multiple tenant onboarding and diverse service management. We perform a substantial case study of an online conference management system.

1. INTRODUCTION

The emergence of Cloud Computing has been noticed by one and all in the Information and Communication Technology (ICT) industry. Companies are increasingly adopting this new paradigm where they do not wish to commit resources for engineering computing infra-structure. Instead, they acquire these resources as and when they need it as services. Indeed there is increasing focus on so-called Software-as-a-Service [17] (SaaS). This paper aims to analyze the technical issues behind software-as-a-service.

Informally, SaaS has been defined as software deployed as a hosted service and accessed over the internet, without the need (for users) to deploy and maintain additional on-premise IT infrastructure. The move from on-premise license-based software usage, the traditional way software has been delivered to customers for decades, to the centrally hosted, subscription-based SaaS model, is a huge change for software vendors and customers, offering unique business benefits but also several technical challenges. The ownership of the software now truly shifts to the provider, as does the responsibility for providing technology infrastructure and management. The customer organization is spared the burden of purchasing and maintaining server hardware for the applications. From the vendor's perspective, the additional costs of hardware and management are more than offset by leveraging the economies of scale that arise from being able to serve a very high number of customers.

One of the key technical issues behind SaaS lies in it being a shared, centrally hosted software service. In contrast to earlier attempts of application service provisioning, the SaaS delivery model is focused on exploiting economies of scale by offering the same instance of an application (or parts thereof) to as many customers as possible. SaaS vendors support different customer needs from the same common base (application, infrastructure) through appropriate sharing and customizations (termed multi-tenancy in SaaS parlance). Multi-tenant aware applications thus allow providers to provision the hardware and software stack for the application once and run multiple customers on the same infrastructure. Salesforce.com, a popular SaaS CRM application, for example, allows users to customize the data-fields used in the application as well as adding and modifying workflows. This is done by configuration meta-data that is specific for each tenant.

The functionality and quality that different tenants require from a SaaS application might differ. Therefore, a multi-tenant 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. Besides functional requirements, tenants may also have different performance expectations from the SaaS system. This introduces additional complexities, since the infrastructure (application, system) is shared across many tenant organizations, each of which may have a very large number of SaaS sessions running at any point in time. In addition to variations that arise from different tenant needs, the SaaS vendor itself may sometimes want to offer multiple variants of a particular service for different classes of tenants. These variants may offer different grades of functionality / service guarantees at appropriate pricing options. Thus, SaaS vendors have two objectives which need to be balanced while deploying the service infrastructure. Firstly, to attract more tenants, the vendor needs to cater for the varying needs by providing efficient means for tenant-specific adaptation. Secondly, the vendor needs to make sure that the variants of the services retain enough commonalities to exploit the economies of scale attained. Managing variability and deploying commonality in an efficient fashion is a key objective in a SaaS framework.

In this paper, we present a formalization of the SaaS service infrastructure. A SaaS component consists of a set of services $\mathcal{S}$ and a set of tenants $\mathcal{T}$ that interact with the SaaS component to exercise the various features. Each service is a tuple consisting of (a) a collection of features that the service offers for its tenants, and (b) a set of invariants that must be preserved over all executions in any SaaS session across all feature invocations and executions. Each feature is specified as a contract capturing what it offers and terms of using it. Formally, each feature is characterized by a set of pre-conditions, a set of post-conditions and a default method implementation provided by the vendor. The method body is not visible to tenants, and is internal to the SaaS implementation. Intuitively, if a feature provides a certain functionality, it may:

- Expect a certain condition to be guaranteed on onboarding by any tenant who wishes to use it: the feature **precondition** is an obligation for the tenant, and a benefit for the vendor (the feature itself), as it frees it from having to handle cases outside of the precondition.
- Guarantee a certain property on end of feature execution: the feature **postcondition** is an obligation for the vendor, and obviously a benefit (the main benefit of calling the service) for the tenant.
- Maintain a certain property, assumed on entry and guaranteed on exit: the service **invariant**.

A feature can be refined by relaxing the terms of use and / or restricting the expectations while still preserving the invariants. As a result, tenants can demand different variants of a feature which weaken preconditions (but not strengthen them) and strengthen postconditions (but not weaken them) while still preserving the invariants. Each tenant who wishes to access the existing SaaS infrastructure has its own set of requested services. The set of requested services may either be directly available from the SaaS component or may be a variant of the ones under offering. We propose a well-defined semantic foundation for onboarding a new tenant into the existing service infrastructure. A new tenant is allowed to connect to the existing service framework if his request profile is in conformance to the existing service support. When a new tenant is on-boarded, the tenant will study the existing SaaS model and may request a new variant of a feature (or even, a new feature). If it is a legal variant, then it may be absorbed in the variant set for the feature (*feature variant expansion*), else this has to be represented as a new feature (feature set expansion) or even a new service (service set expansion).

The adoption of a formal foundation provides a simple elegant platform for defining and analyzing the SaaS components. Given a collection of available services with descriptions of their dynamic behavior, and a new tenant with its request profile of a set of service requirements, we develop an automatic mechanism for determining whether the new tenant can be onboarded. In this work, we show how the onboarding question can be easily answered if the services are given as in our intuitive formalism. We present a substantial case study of a practical SaaS system to motivate our method.

Organization. This paper is organized as follows. In Section 2, we present an intuitive SaaS application. Section 3 presents the formal model for a SaaS service application. We show how to utilize the proposed model for application management including service commonality measurement and tenant onboarding selection. A detailed case study is presented in Section 5, followed by some discussion on how our model can facilitate a SaaS provider to design his offerings in a structured manner in Section 7. Finally, Section 6 summarizes a survey of relevant work in this domain, and gives the concluding remarks of this paper.

2. A WORKING EXAMPLE

In this section, we present a fragment of an online conference management system as a representative SaaS component. The example has been adapted from the COMS conference system [1]. The web-based conference management software handles registrations, paper submission, the review process and the selection of papers. Its numerous features combined with the high level of control given to organizers makes it a powerful, yet flexible system. Below, for the purpose of explanation, we take two representative services along with the supported features to explain our formalism. We will present a detailed case study in a later section.

- Author services (S1): This consists of the following features.
  - Author registration (A1)
  - Abstract submission (A2)
  - Paper Submission (A3)
  - Paper Modification (A4)
- Reviewer services (S2): This consists of the following features.
  - Reviewer registration (R1)
  - Review management (R2)
  - Automatic Progress Tracking (R3)
  - Review submission (R4)

We present below an informal implementation description of the features in an intuitive syntax popular in the Design-by-Contract (DBC) [12] community. We will explain the detailed syntax later.

Author Services

Registration (Input: AuthorDetail = <Name, Email>) : $A1_0$
Require
Valid email address and non-empty AuthorName
Do
// Actual implementation goes here
Ensure
Automatically generated password emailed to author

AbstractSubmission (Input: Email, Abstract) : $A2_0$
Require
1. Valid email address
2. Author must have been registered
Do
// Basic authentication
// HTML editor allows formatting of the abstract online
Ensure
Unique paper ID emailed to author

PaperSubmission (Input: Email, Paper) : $A3_0$
Require
1. Paper in PDF format
2. File Size less than 1 MB
Do
// Basic authentication

// Actual implementation goes here
Ensure
Paper Submission status displayed

PaperModify (Input: Email, Paper ID) : $A4_0$
Require
Do
// Basic authentication
// Actual implementation goes here
Ensure
Paper Modification Success / Denial displayed to author

Reviewer Services

Registration (Input: ReviewerDetail = <Name, Email>, ConflictsOfInterest, TopicAreas) : $R1_0$
Require
Valid email address and non-empty Name
Do
// Actual implementation goes here
Ensure
Automatically generated password emailed to reviewer

ReviewMgmt (Input: Papers, Reviewers) : $R2_0$
Require
Each reviewer is a registered reviewer
Do
// Actual implementation goes here
Ensure
1. A reviewer is not assigned his own paper
2. A reviewer is not assigned papers of authors in conflict list

AutomaticProgressTracking (Input: ReviewerDataBase) : $R3_0$
Require
Do
// Actual implementation goes here
Ensure
Reminder Email sent to all assigned reviewers

ReviewSubmission (Input: Email, Password) : $R4_0$
Require
Do
// Basic Authentication check
// Review submission
Ensure
Review Submission Success / Denial displayed

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

3. FORMAL MODEL

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 to make our formal model amenable to model checking). 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. 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 $meth(F) \subseteq G_S \times G_S$, that is, we allow some non-determinism in the formal model corresponding to a single service. For convenience, 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.

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

Furthermore, we can also bring in the concept of pre-condition and post-condition of a service by augmenting the above constraint as follows

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

Examples of service and feature invariants of the COMS example presented in Section 2 are

- **Invariant for author services S1:** All features provided in this service can be only executed by the contact author of the paper.
- **Invariant for paper submission feature A3:** Paper submission must be completed before the deadline
- **Invariants for reviewer services S2:** A review feature cannot modify / delete reviews from the review database.
- **Invariants for reviewer registration feature R1:** Only authorized email account can be used to register as a reviewer.

3.1 Feature Variants

Services thus defined can be refined in two ways:

- Feature expansion - Modification of an existing feature.
- Feature set augmentation - Addition of new features.

The concept of feature variant comes from the fact that the variation may come from the tenant in the feature implementation / terms of use or even the expected outcome of the feature. This is a very common situation since usually the vendor will offer one concrete implementation to start with for each feature, while tenants can demand different variants while still having to meet the invariant, or have a new (different) feature altogether – which may be considered to be more costly than a variation. In all the cases, we assume the service invariants have to be preserved. Some situations where we may have such a need are as follows:

1. Relaxation / Restriction of Terms of Use: This is a very common requirement since a new tenant may not be agreeable to the existing pre-condition of a service invocation and may demand a relaxation that he can conform to. In some cases, 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, or may wish to have some more out of the feature execution.

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 present a concrete classification of the above scenarios using the notion of refinement and variant.

DEFN 1. **[Condition restriction:]**
Let C_i and C_i are two sets of conditions (quantifier-free first-order logic formulas over variable set $\overline{V_i}$ and $\overline{V_j}$), $C_i \prec C_j$ denotes C_j is a restriction of C_i if and only if

- $\overline{V_j} \subseteq \overline{V_i}$, *and*
- $\{c_j^v \Rightarrow c_i^v | \forall v \in \overline{V_j}\}$, *where c_i^v and c_j^v are the conditions in C_i and C_j on the same variable v, respectively. In other words, C_j is a set of* stronger *conditions that must hold in order for C_i to hold.*

DEFN 2. **[Feature variant and refinement:]**
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 a variant of feature F if there exists a variant V

$$\{v = \langle pre(v), meth(v), post(v), \overline{\Psi_F} \rangle | v \in Var(F)\}$$

such that v and s are bound by the same *service and feature invariants, and any of the following four cases holds (with at least one inequality strict in each cases)*

1. $pre(s) \preceq pre(v) \wedge post(s) \preceq post(v)$
2. $pre(v) \preceq pre(s) \wedge post(s) \preceq post(v)$ *(v is a* **refinement** *of s)*
3. $pre(s) \preceq pre(v) \wedge post(v) \preceq post(s)$
4. $pre(v) \preceq pre(s) \wedge post(v) \preceq post(s)$

In any of the listed four situations, s is considered as a variant of feature F. In particular, the second type of the feature variant corresponds to the case where an existing feature variant v is a **refinement** of new service request s. In the context of contract refinement, if we have a new client of service request s, it will be a valid client of its refinement, i.e., it can legally call the refined method ($meth(v)$) and in turn receive (at least) what it expects. In both the cases, the new service request F' can be supported with some sharing of existing code base / application infrastructure. In case of a refinement, no new feature variant is created. In the case of a variation, a new feature variant is created for F' which is then absorbed in the variant set of the feature F (*feature expansion*) of which it is a variant.

Each feature F in the system has a list of variants attached to it. Initially, each feature has a default variant (subscripted as 0 in our working example) which corresponds to the default variant offered by the SaaS vendor. As the system matures, new requirements are introduced and the variant set is augmented with the variations.

On the other hand, in case none of the conditions above hold, we create a new feature implementation for s. This is called *feature set augmentation* (Section 4). Note that we require a strong condition for feature expansion, i.e., a new service request is supported as feature variant only if it can be obtained by weakening/strengthening pre/post conditions of existing variants of a feature. Consequently, we may have to create a new feature in some situations, even the new request is bound by the same service and feature invariants. This gives the SaaS provider an opportunity to merge some features during the evolution of the SaaS application when new tenants are onboarded (to be discussed in Section 4.5).

3.2 Feature Variant Examples

We now explain the concepts on our conference management system example presented in Section 2. Following is an example of a new feature in the author services (S1).

New Feature in S1

RebuttalSubmission (Input: Text, Paper ID) : $A5_0$
Require
Rebuttal text is less than 200 words
Do // Actual Implementation
Ensure
1. Rebuttal Emailed to PC members
2. Successful submission feedback sent to author

Following are some possible variants of the feature abstract submission ($A2$) in the author services (S1).

Example Feature Variants

AbstractSubmission (Input: Email, Abstract)
$A2_1$ **(variant of** $A2_0$**)**
Require – *Weaker precondition*
Valid Email
Do // Modifying the implementation from $A2_0$
1. Remove checks and code related to author registration.
2. Add functions for password generation and distribution.
Ensure – *Stronger postcondition*
1. Unique paper ID emailed to author
2. Author is sent an automatically generated password

AbstractSubmission (Input: Email, Abstract)
$A2_2$ **(variant of** $A2_1$**)**
Require – *Stronger precondition*

1. Valid Email
2. Text is less than 500 words

Do // Modifying the implementation from $A2_1$

1. Perform additional check on abstract text size
2. Remove functions related to the password requirement.

Ensure – *weaker precondition*

Unique paper paper ID emailed to author

---

4. 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 services 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 services 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 there exists a feature F in $\mathcal{S}$ such that s is a feature variant of F (DEFN 2), s can be added into the SaaS system as a variant of feature F, by modifying the implementation of the existing variant in F.

- **[Feature set augmentation :]** If there exists a service S in $\mathcal{S}$ such that s is bounded by the same invariants $\overline{\Psi_S}$ with S. However, s is not a variant for any existing features $F \in S$ (due to unjustification of the defined pre/post condition relations, or s is bounded by different feature invariants). As a result, s can be added into the SaaS system as a new feature of service S. A new feature implementation is required to onboard this tenant.

- **[Service set augmentation :]** If s is not bounded by the same invariants with any existing service in $\mathcal{S}$. In this case, the tenant requires a new service.

In all the cases above, the question of on-boarding depends on the SaaS application provider. If the onboarding requires supporting a new service feature or a new service, the SaaS application needs to undergo change. This is replete with many design issues, starting from minimally modifying the existing SaaS application to redesigning the service pool to optimize variability. In this paper, we propose a decision model with two dimensions, namely the onboarding profit and level of system reuse, to help SaaS providers with the new tenants onboarding problem.

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

4.1 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 $V(t)$. As we have discussed, $V(t)$ depends on the charging mechanism of the SaaS system. In this section, we present a cost model to facilitate SaaS providers to decide the cost of onboarding a new tenants. For now, we consider only the *salary* cost, which the SaaS provider have to pay the system developers to make necessary modifications to onboard a new tenant t (on a human-hour basis). Intuitively, the more complex 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 actually 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):]** In this case, the new request s is added as a variant of existing feature F, and the implementation can be derived from an existing variant with minimal modifications. Let v be the existing variant in F that satisfies one of the four constraints with s in DEFN 2. To enable a quantitative cost model, we assume a constant cost value with every weakening/strengthening of a precondition/postcondition as follows.

  - **[Weaken postcondition cost (WPostC):]** In case the new request s requires a weaker postcondition than v, implementation of v can still be used with no modification.
  - **[Strengthen precondition cost (SPreC):]** To support a stronger precondition in the new request service s, only corresponding check needs to be performed before invoking an existing implementation of v ($meth(v)$). No changes is required in the method body.
  - **[Weaken precondition cost (WPreC):]** If the new request s has a weaker precondition, some assumptions or assertions in the original implementation of v may be violated. As a result, the developer can still derive implementation of s from implementation of v, by removing certain checks and modifying the code accordingly.
  - **[Strengthen postcondition cost (SPostC):]** If the new request s requires a stronger postcondition, the developer needs to add more code in the implementation of s to support the new functionality.

  The above-mentioned modification costs of a single condition are arranged in increasing order of cost. Let $C1$ and $C2$ are two set of conditions, and $dist(C1, C2)$ denotes the

number of conditions in $C1$ that do not occur in $C2$, the cost $FEC(s, v)$ of supporting the new request s by modifying the implementation of its existing variant v can be calculated as follows.

$$FEC(s,v) = \begin{cases} dist(pre(v), pre(s)) \times WPreC + \\ \quad dist(post(v), post(s)) \times WPostC \\ \quad \text{if } pre(s) \preceq pre(v) \wedge post(s) \preceq post(v), \\ dist(pre(s), pre(v)) \times SPreC + \\ \quad dist(post(v), post(s)) \times WPostC \\ \quad \text{if } pre(v) \preceq pre(s) \wedge post(s) \preceq post(v), \\ dist(pre(v), pre(s)) \times WPreC + \\ \quad dist(post(s), post(v)) \times SPostC \\ \quad \text{if } pre(s) \preceq pre(v) \wedge post(v) \preceq post(s), \\ dist(pre(s), pre(v)) \times SPreC + \\ \quad dist(post(s), post(v)) \times SPostC \\ \quad \text{if } pre(v) \preceq pre(s) \wedge post(v) \preceq post(s) \end{cases} \quad (1)$$

Finally, in case there are more than one existing variants of s, the developer can choose the support s from the *nearest* existing variant with the minimal feature expansion cost, i.e.,

$$FEC(s) = \min \forall v \in Var(F)\{FEC(s, v)\}$$

where s is a variant of F and $Var(F)$ is the set of all existing variants of F.

- **[Feature set augmentation cost (FAC) :]** This is the cost of developing and provisioning a new feature.
- **[Service set augmentation cost (SAC) :]** This is the cost of developing and provisioning a new service.

The cost of onboarding a tenant is an estimate of 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

$$cost(t, \mathcal{S}) = \sum_{\forall s \in requires(t)} DOC|FEC(s)|FAC|SAC \quad (2)$$

where DOC, FAC and SAC are the constant cost of direct onboarding, feature set augmentation and service set augmentation, respectively; and $FEC(s)$ is the cost that implements s from its nearest variant in the existing SaaS system. Consider the following example.

EXAMPLE 1. Consider our example SaaS application for the conference management system. Let us assume initially only the default feature variants as described in Section 2 are on offer. Let us consider the cost of onboarding 4 tenants T1, T2, T3 and T4 with the requirements as below:
requires(T1) = $\langle S1'\rangle$ where $S1'$ = $\langle A1_0, A3_3, \overline{\Psi_{S1}}\rangle$
requires(T2) = $\langle S1'', S2\rangle$ where
$S1''$ = $\langle A1_0, A2_1, A3_4, A5_0, \overline{\Psi_{S1}}\rangle$
requires(T3) = $\langle S2\rangle$
requires(T4) = $\langle S1''', S2'\rangle$ where
$S1'''$ = $\langle A1_0, A3_1, A4_0, \overline{\Psi_{S1}}\rangle$ $S2'$ = $\langle R1_0, R2_2, \overline{\Psi_{S2}}\rangle$
The onboarding cost associated with onboarding T3 is lowest as compared to the cost of onboarding the other tenants. □

4.2 Commonality considerations

An important factor that characterizes the level of reuse of a SaaS system is the notion of *commonality* of a service. Commonality is defined as the degree of sharing of features by tenants in a SaaS system. We expect that for each feature, a significant fraction of tenants should subscribe to it (including the variants). To compute the overall commonality of a service in a SaaS component, we measure for each feature, the fraction of tenants subscribing to it (including its variants), and then take the average over all features. Let N^f denote the number of tenants subscribing to feature f (either the default one or some variant of it). Let $|F_S|$ denote the total number of features available as part of service S including default and added variants of each feature. Let T_S (> 0) denote the set of tenants subscribing to service S.

DEFN 3. **[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 is given by:

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

EXAMPLE 2. Consider our example SaaS application for the conference management system after onboarding tenants T1, T2, T3 and T4. Based on the requirements of these tenants, the initial SaaS system is augmented with 1 variant for feature $A2$, 2 variants for feature $A3$, 1 variant for feature $R2$ and a new feature $A5$ (with a default variant). Therefore, S1 has 5 features and 3 tenants subscribing to it. S2 has 4 features and 3 tenants subscribing to it. The degree of commonality of service S1 is: $\frac{3+1+3+1+1}{3*(5+1+2)} = \frac{3}{8} = 0.37$. The commonality of S2 is: $\frac{3+3+2+2}{3*(4+1)} = \frac{2}{3} = 0.67$. □

Commonality is an important consideration from the perspective of the SaaS vendor. A high value of commonality indicates that on an average, there is a high percentage of feature sharing. The SaaS vendor would always like to have an improvement in commonality as new tenants are onboarded. However, this may not always be the case and commonality may decrease as well. In general, the SaaS vendor would like to ensure that the commonality of each service remains above a certain threshold. We call this the **commonality threshold**.

4.3 Cost and Commonality Considerations

When a new tenant wants to onboard an existing SaaS system, the following factors need to be considered:

- Net Profit
- Changes in commonality

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. Consider the following example.

DEFN 4. **[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 $COMM$.

4.4 Generic Tenant Onboarding Problem

In Section 4.3 we consider the onboarding activity for a standalone tenant. In the general case, there might be multiple tenants waiting to be onboarded. More than one tenant might request for a new feature, which is associated with a prohibitive development cost and not a prudent decision to support if tenants are considered in isolation. However, considering the tenants collectively, the net profit might be increased though there is a drop in net profit if we consider the onboarding activity for each of the tenants separately. This gives rise to an optimization problem for onboarding tenants.

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

$$\sum_{t \in \hat{T}} V(t) - \sum_{t \in \hat{T}} cost(t, \mathcal{S}') \tag{3}$$

where $cost(t, \mathcal{S}'$ is the cost of onboarding t in the new system $\mathcal{S}'$ as calculated in Equation 2. As discussed in Section 4.1, the cost of supporting a new requirement depends on how this new requirement can be implemented. However, when a set of new tenants is to be onboarded, the cost of supporting a new requirement s is not only determined by the existing SaaS system. Because s can also be accommodated from other new requirements by $\hat{T}$. As a result, we should consider the resulting new SaaS services $\mathcal{S}'$ when deciding the cost of s.

In the initial stages of the SaaS system, the SaaS vendor tries to onboard as many new tenants as possible, to maximize the revenue, even at the cost of supporting new features and services. In steady state (existing SaaS system with many onboarded tenants), the SaaS vendor will also want to consider improvement in commonality along with net profit. This gives rise to a biobjective optimization problem of tenant subset selection as below.

DEFN 5. **[Profit-Commonality Maximizing tenant subset:]** Given a set of potential tenants $t_1 \ldots t_k$, each characterized as a tuple $\langle requires(t_i), V(t_i) \rangle$ and a commonality threshold $COMM$ on an existing SaaS system $\langle \mathcal{S}, T \rangle$, the onboarding problem involves selecting the tenant *subset* $\hat{T}$ 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 [8, 19].

It may be noted that in our formal model, the commonality measure is defined at the service level, thereby giving it a vector perspective. It is easy to lift the commonality to the SaaS component level by taking the average service commonality. In that case, the optimization problems can be easily redesigned to reflect the scalar nature of the commonality measure.

4.5 Merging Existing Features

As discussed in Section 3.1 we require a relatively strong condition for feature expansion. Our framework allows a new request to be supported as feature variant only if it can be obtained by weakening/strengthening pre/post conditions of existing variants of the feature, which ensures the correctness of implementation reuse as in our cost model.

As a result, the SaaS system at certain state may have two or more features that are bounded by the same feature variants. These features may also contain variants that can also be treated as variants of other features. In such scenario, the SaaS provider can merge these features together as one single feature, which increases the maintainability of the system and also helps the decision problem for future tenants onboarding.

Consider the following situation. Let $v0$ be the only variant of feature F in the current system, where

- $pre(v0) = p1 \wedge p2$, and
- $post(v0) = q1$

Assume a new tenant requests a service s where

- $\overline{\Psi_s} = \overline{\Psi_F}$, and
- $pre(s) = p1 \wedge p3$, and
- $post(s) = q1$

In our framework, s will be supported as a new feature $F1$ (feature set augmentation), since it is not a variant of any existing variants of F. It is due to that supporting s directly from modifying $meth(v0)$ may not be very desirable. However, assume another new tenant requests a service s' where

- $\overline{\Psi'_s} = \overline{\Psi_s} = \overline{\Psi_F}$, and
- $pre(s') = p1 \wedge p2 \wedge p3$, and
- $post(s') = q1$

s' can be supported as variant of either F or $F1$ (it holds a stronger precondition than any existing variant in F and $F1$). Suppose we generate the implementation of s' from $v0$, now s in $F1$ is also variant of s' in F. We can then merge these two features as a single feature that contains all three above-mentioned variants.

After onboarding a tenant (or a set of tenants), the SaaS provider may perform a merge operation on the resulted system. Formally, two feature $F1$ and $F2$ should be merged into a single feature (which contains all variants in both $F1$ and $F2$) if

- $\overline{\Psi_{F1}} = \overline{\Psi_{F2}}$, and
- $\exists v1 \in F1, \exists v2 \in F2$, s.t. $v1$ is a variant of $v2$.

5. CASE STUDY

In this section, we present a case study of the Conference Online-Management System (COMS) [1].

5.1 COMS System

The COMS offers various services to manage different users and events of a conference. To provide flexibility, COMS almost all services/features are optional and can be configured by conference administrator. Following are the services on offer from COMS.

1. **System User Services:** This has the following features.
   (a) register with the system, create a user account

(b) retrieve lost passwords

(c) log-in and view welcome page

(d) change the interface language (if COMS is configured in multilingual mode)

(e) review and maintain personal data

(f) book items (conference registration, workshops, accommodation, etc.)

(g) view the status of booked items

(h) view the payment status

(i) submit a conference paper and thereby become an author

(j) conduct duties as assigned by the Chair (Administrator, Editor, Reviewer, Clerk)

2. **Chair services:** This has the following features.

(a) configure all system features

(b) enable one or several of the interface languages (monolingual or multilingual mode)

(c) set time windows for registration, paper submission, peer reviewing, etc.

(d) nominate assistants (Administrator, Editor, Reviewer, Clerk)

(e) configure the welcome page, post current news

(f) configure templates for automatic confirmation emails

(g) communicate by bulk email with pre-determined groups of users

(h) view the log-file of all sent bulk email

(i) view and edit the details of all participants

(j) create new user accounts

(k) delete existing user accounts

(l) view and change the booked items of any participant

(m) view and change the payment status of any participant

(n) view, edit or delete any submitted abstract

(o) download the files of submitted papers by FTP

(p) create and view administrative notes for any participant that are not visible to participants

(q) assign tasks to editors manually or automatically

(r) view the results of the reviewers' bidding for papers

(s) assign tasks to reviewers manually or automatically (optionally based on a bidding process)

(t) view statistics about participants and paper submissions

(u) track progress of the assigned edit and review tasks

(v) select or reject papers manually or automatically, based on the results of peer reviews

(w) assign presentation types (oral, poster, etc.) to accepted papers

(x) distribute accepted papers over conference sessions

(y) create a draft of the book of abstracts automatically

(z) export database contents in different formats for use with other applications

3. **Reviewer services:** This has the following features:

(a) view all submitted papers

(b) bid for papers (state preferences which papers one would like/not like to work on)

(c) view details of authors (unless blind reviewing is in force)

(d) view list of assignments, filter and sort by several criterions

(e) conduct reviews

(f) view who else is assigned to review a given paper

(g) view the appraisals by other reviewers

(h) receive reminders and other bulk email messages from the Chair that are specifically targeted at Reviewers

4. **Editorial services:** This has the following features.

(a) view list of assignments, filter and sort by several criterions

(b) view and edit details of authors

(c) view and edit details of submissions

(d) receive reminders and other bulk email messages from the Chair that are specifically targeted at Editors

5. **Author Services:** This has the following features.

(a) submit conference contributions

(b) view submissions and edit them online

(c) withdraw submissions

(d) upload files of formatted papers

(e) replace uploaded files by updated versions

(f) submit keywords with each paper

(g) propose a type of presentation (oral, poster, etc.)

(h) propose the conference session for a presentation

(i) view the results of peer reviewing

(j) receive reminders and other bulk email messages from the Chair that are specifically targeted at Authors

6. **Participant Services:** This has the following features.

(a) register for the conference and book other items (conference dinner, excursions, accommodation, etc.)

(b) view the payment status

(c) view and print an invoice, bill or receipt of payment

(d) pay conference fees online

(e) receive conference news and other bulk email messages from the Chair that are specifically targeted at Conference Participants

5.2 Experimental Results

To illustrate the use of our proposed multi-tenant SaaS model, we choose the authors service shown in Section 5.1 (Service 5) as a case study. For each feature F offered in this service S ($F_S = \{F_a, \ldots, F_j\}$), we assume several possible pre and post conditions for it. As a result, by choosing different combinations among them, we are able to build a diversity of variants of each feature (refer to

	DOC	$WPostC$	$SPreC$	$WPreC$	$SPostC$	FAC
$set1$	20	50	100	180	200	2000
$set2$	20	100	120	160	300	1600

Table 1: Values for constant cost types in Section 4.1.

Section 3.1 for examples on variants of feature F_a –"submit conference contributions"). Note that the pre-conditions, implementations, and post-conditions may be correlated (e.g., certain post-condition is invalid for a particular pre-condition or implementation), which restricts number of different variants can be created for a feature.

A tenant $t = \langle requires(t), V(t)\rangle$ randomly subscribes one or more features in F_S. For each required feature $F \in requires(t)$, we randomly select one or more pre and post conditions, subject to the pre-defined correlations. An initial system $\langle \mathcal{S}, T\rangle$ is created with 100 such randomly generated tenants ($|T| = 100$). In the experiment, we show how to onboard another 20 randomly generated potential new tenants $\hat{T}$ ($|\hat{T}| = 20$). As discussed in Section 4, the problem is to identify a subset of tenants $\hat{T}' \subseteq \hat{T}$, such that the resulted new system $\langle \mathcal{S}', T'\rangle$ is Pareto optimal w.r.t profit and commonality, where

$$\mathcal{S}' = \mathcal{S} \bigcup_{t \in \hat{T}'} requires(t), \quad T' = T \cup \hat{T}'$$

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 (assumed be to 200 in our experiment). For each tenant $t \in \hat{T}'$, the cost of supporting t in the new system S' can be calculated with Equation 2 in Section 4.1. DOC is the direct onboarding cost when (i) a new tenant requires a feature/variant that has already been supported in the initial system $\mathcal{S}$; or (ii) more than one new tenants require the same feature/variant (only one implementation cost should be considered, the rests can be directly onboarded from that implementation). $WPreC, SPreC, WPostC$, and $SPostC$ are the costs of weakening/strengthening a single pre/post condition, in order to build a new implementation for a newly required variant. FAC is the cost of adding a new feature in the system. In our experiment, we assume two sets of constant values for each type of cost, which are shown as $set1$ and $set2$ in Table 1. Note that we assume $WPostC < SPreC < WPreC < SPostC$ to approximate the real world programming efforts of creating a variant from existing implementation. However, it is not an assertion that requires to hold in our tenant onboarding cost model. Different values can be given to each cost type according to the real situation. For now, 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 each subset, the profit of onboarding $\hat{T}'$ can be calculated with Equation 3 in Section 4.4. In our experiment, the absolute profit is correlated to the constant values assumed for RPS (revenue per subscription). and different implementation costs. Clearly, a lower RPS or higher implementation costs will lead to less profit for the SaaS provider. The commonality of resulted new system can be calculated as in Definition 3. We maintain a set of Pareto optimal solutions, such that there is no other subsets, if onboarded, can 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

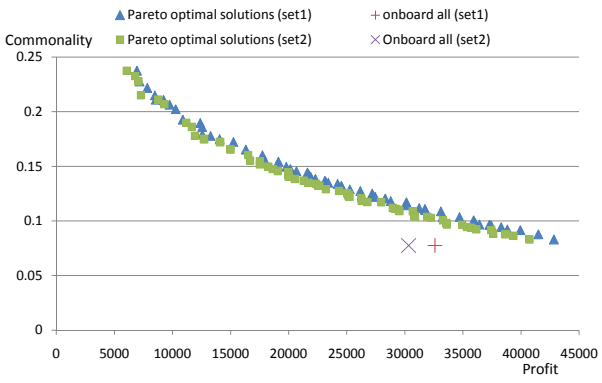

Figure 1: Profit and commonality by onboarding new tenants.

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, for a larger number of onboarding tenants to be considered, evolutionary algorithms ([8]) can be implemented for a more effective Pareto front computation.

Figure 1 shows our experimental results for the above-mentioned system setup. The triangle marks represent all the Pareto optimal solutions for the given cost values as $set1$ in Table 1. Each Pareto optimal solution is associated with the corresponding subset of tenants to be onboard. 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 1 also shows a dominated solution corresponding to onboard all the 20 tenants in $\hat{T}$ with the implementation cost $set1$ ("onboard all ($set1$)"), with $Profit = 32600$ and $commonality = 0.0775$ (labeled with the $+$ mark). Clearly, it is not a good design choice and should be avoided. Similarly, Figure 1 also shows the Pareto optimal solutions and a dominated solution (onboarding all 20 tenants) for the cost values of $set2$ in Table 1. We can observe quite similar trends in the two sets of computed Pareto optimal solutions. It shows that given the same initial system and tenants to be onboarded, the tenant selecting problem is not very sensitive to the exact modification cost required to support the new tenants (as long as the relative relations between the costs keep unchange). In other words, our experimental results show that if a subset of tenants is considered to be a good candidate to onboard to the current system, it is very likely that onboarding the same subset of tenants under a different cost scheme will also result high profit and good commonality.

The commonality values obtained in this experiment is relatively low (between 0.07 to 0.24). It is due to we consider a large set of possible variants for each feature in the author services (in average, 12 variants are created for a feature), in order to highlight the trend between profit increasing and commonality decreasing. In particular, we treat two subscriptions as two distinct variants even if they have very subtle differences in their requirements. We also assume each tenant may randomly subscribe one of the possible features. In reality, a SaaS system usually has a higher commonality degree, since many tenants will agree with the same (or very close) service contract.

Finally, in this experiment, we use only one single service (the author services in the COMS) to illustrate our framework. However, for a SaaS system with multiple services, the optimization problem can be easily adopted at the system level by considering the average commonality of all services.

6. RELATED WORK

Many existing work on multi-tenant applications focus on shared data architecture and security management [6, 9, 2, 18]. On the other hand, there has not been many research on multi-tenancy SaaS systems from software engineering perspective. Models and techniques successfully employed in software product line engineering ([16]) have been applied in multi-tenancy systems to manage configuration and customization of service variants ([7, 13, 14]). In particular, [14] extends the variability modeling ([3]), which provides information for a tenant to choose/customize the SaaS application and guides the SaaS provider for service deployment.

The work of [4] discusses some potential challenges in implementation and maintenance of multi-tenancy 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 [5]. This involves extending user-authentication mechanisms, introducing tenant-specific software configuration and adding an application layer to extract tenant-specific views from the shared database. The recent paper [15] also studies tenant specific customizations in a single software instance, multiple tenant setup.

The work of [10] develops a multi-tenant placement model which decides the best server where a new tenant should be accommodated. The placement mainly considers the hardware resources including CPU and storage usage. In principle, a new tenant will be placed on the server with minimum remaining residual 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 [11]).

7. CONCLUSION

In this work, we have presented a formal framework for multi-tenant SaaS systems. Our focus here is on the software service and not the entire service infra-structure (such as database, OS, file systems). Even though some existing works (as mentioned in the preceding) have articulated the difficulties in managing diverse tenant requirements in multi-tenant service software — a comprehensive approach is missing. Our paper attempts to fill this gap. We use our formal framework for managing tenants and deciding on tenant on-boarding in a multi-tenant SaaS. This done by multi-objective optimizations from the point of view of the service provider.

In our current framework, we consider a single constant cost for weakening/strengthening each pre/post condition. To allow more accurate and practical SaaS management, a fine-grained cost model can be designed specifically for a given SaaS application. In particular, the pre and post conditions of a tenant requirement can be classified into more detailed categories (e.g., functional and non-functional requirements). Such extensions of our framework to support non-functional requirements remain an important direction of future research.

8. REFERENCES

[1] Conference service mandl. `http://www.conference-service.com/index.html`.

[2] 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.

[3] J. Bayer, S. Gerard, Ø. Haugen, J. Mansell, B. Møller-Pedersen, J. Oldevik, P. Tessier, J. Thibault, and T. Widen. Consolidated product line variability modeling. *Software Product Lines*, pages 195–241.

[4] 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.

[5] 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.

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

[9] 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.

[10] 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.

[11] 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.

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

[13] 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.

[14] 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.

[15] Nitu. Configurability in SaaS (software as a service) applications. In *Proceedings of the 2nd India software engineering conference*, pages 19–26, 2009.

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

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

[18] 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.

[19] 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.