Engineering Multi-Tenant Software-as-a-Service Systems

E.Geetha Rani#1, CH.Suguna Latha*2, D.Anusha#3, M.Satya Sukumar*4 T.Swathi#5

#1Information Technology, Gudlavalleru Engineering College, Gudlavalleru, jntuk, A.P, INDIA

#3Information Technology.PVPSIT,Vijayawada, jntuk, A.P, INDIA

#5Information Technology.Gudlavalleru Engineering College, Gudlavalleru, jntuk, A.P, INDIA

**2Information Technology, Gudlavalleru Engineering College, Gudlavalleru, jntuk, A.P, INDIA*

**4Information Technology,PPDCET,Surampalli(V),jntuk,A.P,INDIA*

ABSTRACT

Increasingly, Software-as-a-Service (SaaS) is becoming a dominant mechanism for the consumption of software by end users. From a vendor's perspective, the benefits of SaaS arise from leveraging economies of scale, by serving a large number of customers ("tenants") through a shared instance of a centrally hosted software service. Consequently, a SaaS provider would, in general, try to drive commonality amongst the requirements of different tenants, and at best, offer a fixed set of customization options. However, many tenants would also come with custom requirements, which may be a pre-requisite for them to adopt the SaaS system. These requirements should then be addressed by evolving the SaaS system in a controlled manner, while still supporting the needs of existing tenants. This need to balance tenant variability and commonality, and to optimize on development and testing effort, can make the evolution of multi-tenant SaaS systems an interesting engineering challenge; this has strong economic undertones as well, given the "pay-per-use" subscription model of SaaS, and the cost of incremental development and maintenance to cater to new tenant needs. In this paper, we outline a set of research issues in the design, testing and maintenance of multi-tenant SaaS systems, and highlight some of the interesting optimization questions that arise in the process. Presenting specific technical solutions is beyond the scope of this paper – instead, our goal is to help shape a research agenda for multi-tenant SaaS that can provide stimulus for further investigation into this area by the software and service engineering research community, and can help advance methodological guidance and tool support for SaaS vendors.

Keywords :

Software-as-a-Service, cloud computing, multi-tenancy, testing, semantics, refinement

1. INTRODUCTION

In recent years, the trend towards "Everything-as-a-Service" (XaaS) as envisioned in Utility Computing's pay-per-use model, has been rapidly gaining ground in the Information and Communication Technology (ICT) world. 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 them as services. Cloud computing, which has emerged as the run-time platform for realizing this vision, may be visualized as a stack of possible service types, ranging from infrastructure-as-a-service (IaaS) at the very base, to platform-as-a-service (PaaS)S, to finally, Software-as-a-Service or SaaS – the main focus of this paper. Informally, SaaS may be described 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. From a SaaS vendor's perspective, the benefits of SaaS arise from leveraging economies of scale, by serving a large number of customers ("multiple tenants") through a shared, centrally-hosted software service. This

translates to lower subscription fees for individual tenants, thereby encouraging entirely new market segments to utilize the benefits of software services – for example, Small and Medium Enterprises (SMEs) who have traditionally been unable to afford steep software license costs, are able to factor in SaaS subscriptions as part of their operational expenses, and thereby give their business the benefit of IT services. For these reasons, SaaS has seen very significant growth over the last few years, and the market outlook for the future continues to be bright. According to a recent IDC report [23], the SaaS market reached $13.1B in revenue in 2009, while the on-premise market shrunk by $7B. The SaaS market is forecasted to reach $40.5B by 2014, representing a compound annual growth rate (CAGR) of 25.3%. By 2014, about 34% of all new business software purchases will be consumed via SaaS [23]. Other industry analysts also share this optimism around Cloud Computing/SaaS e.g. Gartner estimates that over the course of the next 5 years, enterprises will cumulatively spend $112B on SaaS, Paas and IaaS [24]. The business benefits of SaaS notwithstanding, supporting true multi-tenancy in a SaaS system can be very challenging. By true multi-tenancy, we mean a SaaS instance that not only supports the common needs of several tenants, but also the custom requirements of individual tenants to the extent possible. In the traditional mode of on-premise software delivery, or even in the Application Service Provider (ASP) model, each tenant would have a dedicated instance of the base application customized to its needs. However, when several tenants have to share the same application instance in a multi-tenant SaaS, how to handle variations in tenant requirements becomes an interesting question. Clearly, supporting such variations increases the overhead on the SaaS vendor. Also, allowing too much variability can defeat the very purpose of sharing, and make system maintenance very expensive. On the other hand, allowing too little variability may discourage tenants from subscribing to a SaaS in the first place - tenants would be unwilling to compromise too much in terms of changing their business processes to adapt to what the SaaS vendor has to offer. This will be particularly true for many small and medium-sized vendors, who would be less capable to dictate the terms of business engagement with their customers. In fact, industry surveys [25] indicate that the inability to customize SaaS applications to suit their needs is the most significant challenge that customers face with the SaaS offerings they use. In the coming years, this has the danger of slowing down the growth of SaaS beyond those domains where there is little or no need for tenant-specific variations. Such domains may be few in spite of the general move towards industry standards.We believe that for the SaaS paradigm to truly meet its potential, vendors will need to move away from building rigid "one-size-fits-all" systems, or those that offer a fixed set of available customization options from which tenants must select. Instead, *vendors will have to design SaaS systems in a way that allows the applications to evolve with time to cater to the custom requirements of newer tenants looking to onboard the system*. While doing so, vendors should not, of course, lose sight of the end-goal of a shared SaaS – that the commonality amongst tenants remain sufficiently high for a single application instance to be justifiable and viable. Thus multi-tenant SaaS development must involve maintaining this balance between tenant commonality and variability on an ongoing basis, leveraging the benefits of commonality wherever possible, and suitably adapting the design/development/testing/on-boarding process to address the requirements of variability. At its very core, SaaS is a economic model for software consumption, hence much of these activities would have to be grounded on the basis of financial reasoning that can benefit the vendor as well as the tenants. In this paper, we seek to outline a multi-tenant SaaS engineering approach that is motivated by this line of thinking. In particular, we consider the topics of: designing multi-tenant SaaS systems in a way that facilitates reasoning about tenant commonality and variability (Section 4); testing such systems efficiently to avoid redundancies due to shared behaviour while still exercising all points of difference (Section 5); and re-factoring SaaS systems to ease maintenance (Section 6). Elaborating on these issues, we naturally find a set of optimization questions rooted in the SaaS economic model, which can guide decision-making – for example, which set of tenants to onboard, or which subset of services to retire, so that the vendor profitability is maximized, or impact on tenants is minimized. The overall SaaS engineering approach that we outline may be realized through design and analysis toolkits that vendors may use to methodically design, validate, refine and evolve multi-tenant SaaS systems. However, going into specific realization aspects is beyond the scope of this paper – we focus, instead, on the research issues involved and outline possible solution approaches with the hope that this will provide an agenda for further investigation.

2. Related Work

While there is a lot of interest in SaaS in general, we believe that the challenges that arise due to multi-tenancy have not been adequately explored from a software engineering perspective. Much of the existing research on multi-tenant SaaS have focused

on shared data architecture and security management [5, 7, 1, 15], and middleware extensions to address the well-founded concerns due to data/security/isolation. The work of [8] 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. There have also been studies on service performance issues in multi-tenant SaaS [9].

In contrast, there has been relatively little research so far on the impact tenant variability may have on the functionality and evolution of a SaaS system over its lifecycle. This is not surprising given that SaaS is a relatively recent phenomenon, and hence the initial focus is bound to be on issues that are related directly to its feasibility (such as security or performance). However, the fact that a SaaS system needs to functionally cater to multiple tenants is now increasingly understood, leading to research on how to model variability in a SaaS, and how to make a SaaS system more customizable. Models and techniques successfully employed in software product line engineering [14] have been applied in multi-tenant systems to manage configuration and customization of service variants. In particular, [11] extends variability modeling [2], which provides information for a tenant to customize the SaaS application and guides the SaaS provider for service deployment. The work of [3] discusses some potential challenges in implementation and maintenance of multi-tenant systems. It presents an architectural approach which tries to separate the multi-tenant configuration and underlying implementation as much as possible, by adopting the 3-tier 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 [4]. 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. A recent paper [13] also studies tenant specific customizations in a single software instance, multiple tenant setup.In the software product lines community, feature diagrams have been used to capture the similarities and differences between products in a software product family (e.g. see [33] and the references therein). Testing of software product lines described as feature diagrams has been studied in [34], where the goal of test generation is given as the presence/absence of selected features. In comparison, for multi-tenant SaaS systems, we feel it is important to have a holistic view of the commonality/differences across tenants so that it be exploited to sharing of parts of the test suite across tenants.

3.Motivation for this Paper

By and large, the emphasis of the above cited work has been on how SaaS architects may model customization/configuration options through variation points, and make them available to tenants who wish to on-board the SaaS system, so that each tenant may individually decide which set of customization options offered by the vendor to select. However, while a vendor may offer a fixed set of customization options based on its understanding of the domain, we expect that a SaaS – like all other software in the past – will need to evolve based on newer/differentiated capabilities demanded by the users – specially since the user base, spanning across multiple tenants, will be large and diverse. Business imperatives will demand this evolution. One may argue that tenant-specific changes (beyond vendor-offered customizations) go against the very objective of sharing, and that such demands, when they have to be met, should be handled through separate customized instances for individual tenants. However, there is an entire spectrum to be traversed between fully common, shared behavior, to completely different, customized behavior, and we strongly feel that the moot question is not whether tenant-specific changes should be considered, but to what extent they may be accommodated within a single instance, while still retaining the benefits of sharing. From the vendor's perspective, the evolution of a SaaS system due to functional variability amongst tenants raises many interesting questions: how different is a new service variant being requested from the ones that we currently offer? Is it a refinement, or an elaboration of what we have, or will it require significant new development? What impact will it have on the homogeneity of the overall system if we accommodate it? Is the return-on-investment justified? How quickly can we test the changes? At what point does the maintenance overhead of tenant-specific changes start outweighing the benefits due to shared behavior? A service variant we were supporting seems to be having diminishing utility – how do we minimize the impact of retiring it?....and many more. The goal of this paper is to help chart an agenda from the existing work on vendor-driven customizability via variability modeling, to a more tenant-driven evolution of a SaaS system, and the engineering challenges (exemplified by the questions above) that the vendor

has to address to accommodate this evolution. In the rest of the paper, we attempt to elaborate on this agenda.

4.Overview

To motivate the multi-tenant SaaS engineering approach that we outline in subsequent sections, let us consider the following scenario – two major stock exchanges make an agreement to offer joint online trading services for a range of stock transactions (this is a realistic scenario as we have seen from recent news on the Singapore and Australian Stock Exchange [30]). Let us suppose, this has to be made operational very soon to exploit a favorable economic climate.

- Several existing services offered by both the stock exchanges have functional similarities, along with some variations. These services need to be identified and grouped together/merged in the joint trading system to be developed.
- Requirements for several new business services have been elicited with a high number of variants to meet the needs of different financial institutions and grades of investors. There is a need to judiciously invest in new development, so that it generates most value for the ecosystem (stock exchange, customers), while giving priority to the services in most common need.
- As development commences, testing scenarios seem to escalate when considering how each possible tenant is likely to exercise the system. Given the short development cycle, it is critical to reduce testing overhead whenever possible, while still retaining a degree of confidence about the coverage of tenant behavior.
- Post development and testing, the joint trading services are offered and they become very successful. New tenants continue to come on-board and the service and variant portfolio is opportunistically expanded to cater to their needs. At one point, maintenance overhead becomes a bottleneck – somehow the system needs to be re-factored to reduce tenant variability and keep things tractable. These are all realistic scenarios that are likely to occur when a multi-tenant SaaS system – one that tries to maximize tenant commonality while accepting some of their variabilities - is developed and deployed. These scenarios suggest the following topics would be relevant for engineering multi-tenant SaaS:
- *A (Semantic) Model for SaaS Systems*: This will involve modeling the SaaS services and variations, and representing tenant requirements so that they may be mapped to the SaaS system. The model should support semantic reasoning, so that similarities and differences between services and tenant requirements may be analyzed to fine-tune the service model, estimate development costs for tenant requirements and guide tenant on-boarding.
- *SaaS Testing*: Tenants will share many common features, but may also need capabilities that apply only to a subset of other tenants. There is a need to devise efficient test representation and test case generation techniques, so that the testing activity can focus on exercising variations in tenant behavior, and avoid redundancies in testing the common behavior shared across a set of tenants.
- *Re-Factoring SaaS Systems*: A multi-tenant SaaS system may be initiated from customized single tenant instances, whose commonalities need to be merged and variation points accounted for. It will continue to evolve as it accommodates the requirements of new tenants on-boarding the system. Eventually, it may again need to be re-factored – certain service variants may not have a high utility and the vendor may want to retire those while minimally impacting subscribing tenants, while the variability amongst certain sets of tenants may justify separate SaaS instances for them. While the focus of the above discussions has been on functional similarities (or variabilities) between tenants and its implications on the SaaS development cycle, there may also be differences between non-functional requirements (NFRs) of tenants. NFRs may be captured in the Service Level Agreements (SLAs) between the tenants and the SaaS provider, and they will constitute an important element of a multi-tenant SaaS analysis and engineering framework. However, given the orthogonal nature of functional and non-functional requirements and how they may be realized, we restrict ourselves to the functional space in this paper.

5.A Model for Multi-Tenant SaaS

A multi-tenant SaaS system has to be carefully designed to handle the variability that can arise due to the differing needs of tenants. At an abstract level, a SaaS system may be considered as a collection of *services*, where each service in turn, consists of a collection of *operations* that can be invoked by clients. The functionality desired by different tenants out of a service or operation may differ, thereby necessitating support for *variants* of these entities. As the existing literature shows [11], concepts from product-line engineering may be adopted to define variation points to which different variants may be linked, and the variability model may also be used to guide SaaS customization. Moreover, the packaging and deployment of the SaaS may be guided through a set of multi-tenancy patterns that help distinguish

between components that are shared between all tenants or are specific to some tenants [12]. Technically, these constructs provide the basic foundation for supporting variability within a multi-tenant SaaS application architecture. However, to help multi-tenant SaaS systems evolve in a controlled manner, what is needed is not only a way to record different variants, but also to be able to analyze their degree of variability (or conversely, similarity). For example, to onboard a new tenant, its requirements from the SaaS system needs to be mapped onto the set of available services and operations, so that the vendor can determine the gaps that need to be filled through new variants. If a new variant is very similar to an existing service or operation, then the development effort will be relatively small, and the homogeneity of the system will not be impacted too much. On the other hand, if the tenant requires a very different/new type of service or operation, then it may imply significant development overhead, which has to be reviewed not only in light of the potential financial benefits, but also the heterogeneity it introduces and its long-term maintenance implications. In addition to updating a SaaS system to onboard a new tenant, the vendor may also wish to re-factor the system periodically to improve its maintainability (we discuss this in Section 6) – this would also need an analysis of similarities between different services/variants, so that the right decisions may be taken with respect to changes that have to be made to the underlying design.

For these reasons, it would be helpful to enrich the existing variation-oriented modeling of multi-tenant SaaS systems with constructs that enable representation of service semantics. This may be done, for example, using a Design by Contract approach [31], where the semantics of a design entity like a service or operation is captured through the use of pre-conditions, effects/post-conditions, invariants etc. In the SOA world, such a representation has already been explored by the semantic web community to facilitate service discovery, matching or composition, leading to formalisms like OWL-S [32]. We believe that a similar approach can also be taken to establish the semantic underpinnings of a multi-tenant SaaS solution. On top of this, one may define different notions of refinement to understand relationships between services/variants and the ease with which a new variant may be created from existing ones. For example, a variant that only needs weakening of an existing pre-condition may be easier to incorporate than one that introduces a significant new post-condition. Similarly, the addition of a variant of an existing service operation may cause less impact than the definition of a new operation, which is turn, may be deemed to have less overhead than having to define an entirely new service for a tenant. Such an approach would help the vendor estimate the cost of onboarding a new tenant, both in terms of the associated development effort, as well as the degree of heterogeneity that is introduces into the model. The vendor may further define thresholds for this heterogeneity (or conversely, homogeneity or commonality) at different design levels to control and scope the evolution of a multi-tenant SaaS system. Given such a semantic model for SaaS, the onboarding of tenants poses interesting optimization problems. The requirements of a tenant may be represented in terms of services and operations, and we may expect these requirements to be a mix of mandatory (must-have) and optional (good-to-have), which provides a basis for negotiation with the SaaS vendor. Given a tenant's requirements profile, the vendor would like to identify the optimal subset of requirements it should support, so that its net profit is maximized while leading to the best commonality in the resultant system. The vendor's profit would be the difference between the expected revenue from the services/operations based on the tenant's anticipated usage profile, and the cost of additional development, which in turn will depend on the degree to which existing services/operations may be re-used e.g. through refinement. The resultant commonality of the system would reflect the extent to which the services/operations of the updated system are shared between tenants, and the degree of similarity between the variants of a service/operation. Given a set of such tenants to be considered for the next cycle of evolution of the SaaS system, the vendor would be interested to identify the subset of tenants and requirements to support, so that the above profit/commonality criteria are optimized.A variation-oriented semantic model for multi-tenant SaaS can thus provide a sound basis for a controlled evolution of the system.

6.Testing Multi-Tenant SaaS systems

There are (at least) two interesting questions to consider in the area of testing multi-tenant SaaS systems that evolve to accommodate tenant variability: First, when a new tenant is onboarded, how do we test that existing tenants are not impacted by the changes introduced? Second, how do we efficiently test that the SaaS system meets the needs of the different tenants that have been onboarded? In the approach outlined in the preceding section, any new functional capability required by a tenant that is being onboarded, is handled cleanly by defining a new service/operation or its variant. We do not update any existing operation used by current tenants. This ensures that the changes made for the new tenant are isolated, and do not impact the functioning

of the existing tenants. The first question is thus not relevant to our approach, although it will be a core concern for methods that try to overload existing operations to behave differently for different tenants. We do not recommend this since it is likely to result in code that is very difficult to maintain. The second question, however, is very relevant. Given the large degree of commonality that is likely to exist amongst tenants, significant testing resources may be consumed if every tenant has to be fully tested across all applicable scenarios. Rather, we may wish to test only the changes introduced by a tenant. One may argue, of course, that each tenant is different in that it would have its own data set. However, even if tenants are to be comprehensively tested individually, a testing strategy should be devised that exploits similarities amongst tenants to let testers step through the scenarios in a systematic manner. Below, we elaborate on the issues related to multi-tenant SaaS testing, based on the semantic model suggested in the preceding section. We assume a test case to be represented as a sequence of service operation invocations (or it may be relaxed into a partial order). The first issue we consider is test case generation, particularly test cases which do not exist in the current test-suite, but which should be tested once the new tenant(s) are on-boarded. This problem is similar in flavor to the test-suite augmentation problem – where tests are generated to stress program changes, namely executing the changes and propagating the effect of the changes to the program output. The general problem of test-suite augmentation may be addressed via two steps (for example, see [16]). In the first step, a control dependency analysis is done to find a test input to reach/execute the change. Then, in the second step, we modify the path of the change reaching test input to ensure that the program outputs are different with or without the change. One key issue here is to avoid infeasible paths, and for this reason symbolic execution (and path condition calculation) is essential. For multi-tenant SaaS systems, the test-suite augmentation problem will be visualized at a higher level, with the changes defined at operation level. Consequently the individual steps of the analysis (for finding the new tests) will also need to be changed. For reaching the change, we may want to exploit pre-conditions of the operations, instead of performing a fine-grained control dependency analysis. Finally, for propagating the effect of the executed changes, we can analyze the operation post-conditions (along with suitable control flow restrictions) to find a suitable test (in the form of a partial order of operation invocations). Since pre- and post-condition analysis will be central to this method, we envision that symbolic execution will play an important role in the proposed methods. The approach will extend contract-based testing of web services [26, 27]. The second issue relevant to testing multi-tenant SaaS systems is devising a testing strategy that exploits the similarity amongst tenants and structures the test suite accordingly. For this purpose we propose the notion of a *Test-tree*. The root node of a test-tree captures the set of test cases which need to be tested for all the tenants. Each intermediate node of the tree will capture a set of test cases which need to be tested for a subset of tenants. Thus, a partitioning of the tenant set is given by the root-to-leaf paths in the test-tree. To further illustrate the notion of test-tree, we may consider a schematic example.

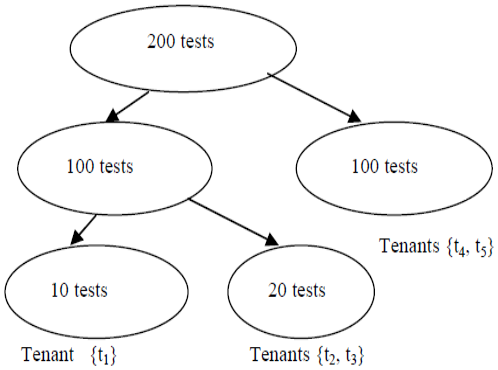

Fig. 1: A Test Tree for Multi-Tenant SaaS

In this example, we have five tenants {t1, t2, t3, t4, t5}. For comprehensive testing, tenant t1's behavior needs to be tested against (200 +100 +10) = 310 test cases. Of these, 10 test cases are unique to t1, hence the SaaS system must be tested on these prior to onboarding of t1. Out of the remaining, 100 test cases are shared with t2 and t3, and 200 test case are shared with all other tenants, so depending on the degree to which these test cases have already been exercised on existing tenants, testers may decide whether to test for a specific case or not. Furthermore, the root-to-leaf paths in the test-tree induce a partitioning of the tenant set – namely {{t1}, {t2, t3}, {t4, t5}}. We feel that the notion of a test-tree is a powerful one, for efficient and systematic testing of multi-tenant SaaS systems. In a broad sense, constructing the test-tree also amounts to a specification of the behavior of tenants in a multi-tenant SaaS system – outlining the similarities and differences across the tenants' usage of the SaaS system. Given such a notion of a test-tree for a multi-tenant SaaS system, we need to study how the tree is modified as new tenants are on-boarded. In this respect, we can be guided by some of the works on software change-impact analysis. Mature tools like Chianti [17] exist for change impact analysis.

Given two program versions, these tools identify the atomic changes (across the two versions) and then find out the tests whose execution is affected by the changes. Such tools are very useful for program understanding, debugging and testing – but from a general software engineering context. For multi-tenant SaaS systems, the atomic changes can be defined more coarsely, possibly in terms of new operations or variants thereof. We can then adapt the works on change impact analysis to find which tests from the existing test suite may be affected, and test-tree transformations have to be defined accordingly.

7. Re-Factoring Multi-Tenant SaaS Systems

There are situations where a vendor may wish to refine a SaaS design, either to improve its maintainability, or to provide better support for multi-tenancy. In particular, we envision the need for three re-factoring techniques that we term *merging*, *splitting* and *pruning*. The goal of merging is to help bootstrap a multi-tenant SaaS design from existing single-tenant ones. Splitting may be used to generate smaller SaaS systems to reduce variability and improve maintenance. Pruning may be used to retire service entities that are of low utility, in a controlled manner to manage impact. We introduce them in the following.

Merging: The merging technique will be useful in moving legacy service systems to the cloud. Imagine a vendor of a SaaS system or an on-premise software product with many instances that have been individually customized and deployed for different customers. The vendor may now want to offer this software on the cloud, and have a single instance shared across the customers, to leverage the benefits of multi-tenancy. From a design perspective, this means that the commonalities and differences across the various customized instances need to be identified and accounted for within a common design – this is where merging comes in. The technique assumes that the individual instances have a SOA-based design in terms of services and operations, and that the semantics of these entities (pre-conditions/effects) is known, or may be discovered by mining the legacy code. Given this, merging will analyze the specification of the different instances to detect similarities in services/operations. Different grades of similarity (from strict to lenient notions) may be used to come up with a merged design that meaningfully groups together similar entities under variation points. The literature on model differencing/merging [17, 18, 19] and semantic web matching [20, 21] will be relevant here.

Splitting: This is the dual of the merging operation. There may be a number reasons why a service provider may want to split a large multi-tenant SaaS system into smaller multi-tenant systems (each system consisting of a subset of services and operation variants present in the original system). For example, it may be due to ease of maintenance. As more and more tenants onboard a SaaS, the service/operation/variant set may keep on increasing. As a result, the software may get bloated, and a direct business consequence of this for the provider would be higher maintenance costs. Secondly, a group of tenants may exhibit similar usage requirements. In such cases, it may make sense to support them out of a separate (smaller) SaaS instance, and maybe charge a higher price for those combinations of services and operations. However, splitting a large multi-tenant SaaS system into multiple smaller ones supporting subsets of tenants, may also lead to some features being replicated across the different instances, and this may lead to new running costs. There is thus a trade-off to be considered. A relevant optimization problem is, given a multi-tenant SaaS system S, divide its tenant set T into K (>=2) non-overlapping sub-sets generating K multi-tenant systems (each system containing all the services/operations/variants needed by its tenants), in a way that leads to maximization of the profit for the SaaS vendor and also leads to the best commonality in the resulting systems.

Pruning: Pruning refers to changes made to a SaaS design by retiring entities (services, operations) that the vendor perceives to be of low utility. This may be based on financial motives. For example, the utility value for a service operation (or a specific variant) may be computed as the ratio of revenue generated from this operation and its running costs – where the revenue is computed over all tenants who have subscribed to the operation, while running costs refer to the cost the provider has to bear to maintain the operation in question (such as cost of associated infra-structure, third party services and so on). We can similarly lift the notion of utility to the level of a service by averaging over all the service operations. When the utility of a service, operation or variant falls below a threshold, the SaaS vendor may decide to retire i.e. withdraw those entities, and thereby save on the running costs. We term this as pruning the SaaS design.

However, retiring a service or operation will impact those tenants who have subscribed to it. If an entire service is retired, the subscribing tenants will lose the associated functionality, and if this represents one of their mandatory requirements, they are likely to leave the vendor, causing revenue loss to the latter. A more controlled way of pruning the SaaS system may be by retiring selected operation variants of low utility, with the plan of offering other variants of these

operations (as substitutes) to the subscribing tenants. Our assumption here is that as long as the vendor is able to preserve a tenant's control flow through the SaaS at the level of the operations invoked, it may still be acceptable to the tenant if certain operation variants are replaced by other suitable variants. Of course, tenants will also need to know the cost implications of this transfer – for example, if the new variants are much more expensive than existing ones - hence the vendor's goal would be to offer those alternative to a tenant that do not result in excessive additional cost. On the other hand, if the provider cannot preserve a tenant's control flow in terms of the operations it needs to invoke, then the tenant may leave the provider. Given this context, the pruning problem may be formulated in terms of determining the subset Sk of low-utility operation variants that may be removed from the SaaS system, such that the number of tenants who may leave is <L, the average transfer costs of remaining tenants is <Q, and the provider's profit is >P, where L, Q and P are suitable thresholds for the respective measures that may be defined by the user/vendor.

8.Summary

Multi-tenancy offers a very attractive proposition to vendors and customers alike, to leverage the economies of scale by sharing a common application instance across many tenants. There is a growing need however, to make multi-tenant SaaS more flexible so that some of the custom requirements of individual tenants can be met even within the shared application instance. Existing approaches try to address this by considering how a vendor may offer a (fixed) set of customization options to tenants, which they can choose from while onboarding. In this paper, we have argued for a more tenant-driven evolution of a SaaS, where a vendor can accommodate changes to a SaaS to meet tenant needs, within reasonable limits. We have then discussed a number of software engineering issues that are relevant to such an evolution, and some of the optimization problems that arise. Specifically, we have considered semantic modeling of multi-tenant SaaS systems, onboarding of tenants with custom needs, efficient testing for multiple tenants with a mix of common/custom behavior, and re-factoring techniques to increase the maintainability and economic value of multi-tenant SaaS systems. We are currently working on formalizing many of the concepts introduced in this paper. This will lay the foundation for a multi-tenant SaaS toolkit with capability patterns for semantic modeling, tenant onboarding, testing and re-factoring, that vendor teams may use to develop, evolve and maintain multi-tenant systems.

9. REFERENCES:

- S. Aulbach, T. Grust, D. Jacobs, A. Kemper and J. Rittinger. Multi-tenant Databases for Software as a Service:Schema- Mapping Techniques. In SIGMOD, pp 1195-1206, 2008.
- J. Bayer, S. Gerard, O. Haugen et al. Consolidated Product Line Variability Modeling. Software Product Lines.
- C. Bezemer and A. Zaidman. Multi-Tenant SaaS Applications: Maintenance Dream or Nightmare? In Proceedings of the 4th Internaitonal Joint ERCIM/IWPSE Symposium on Software Evolution (IWPSE-EVOL), 2010
- C. Bezemer, A. Zaidman, B. Platzbeecker et al. Enabling Multi-tenancy: An Industrial Experience Report. In ICSM.
- F. Chong, G. Carraro, and R. Wolter. Multi-Tenant Data Architecture. MSDN Library, Microsoft Corporation, 2006.
- K. Czarnecki, M. Antkiewicz and C. Kim. Multi-level Customization in Application Engineering. Communications of the ACM, 49(12): 65, 2006.
- 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. Conference on Enterprise Computing, E-Commerce and E-Services (CEC-EEE), 2007.
- T. Kwok and A. Mohindra. Resource Calculations with Constraints and Placement of Tenants and Instances for Multi-Tenant SaaS Applications. In International Conference on Service Oriented Computing (ICSOC), 2008.
- X. Li, T. Liu, Y. Li and Y. Chen. SPIN: Service Performance Isolation Infrastructure in Multi-Tenancy Environment. In Internatonal Conference on Service-Oriented Computing (ICSOC), pp 649-663, 2008.
- R. Mietzner and F. Leymann. Generation of BPEL Customization Processes for SaaS applications from Variability Descriptors. In IEEE International Conference on Services Computing, volume 2, pp 359-366, IEEE Computer Society, 2008.
- R. Mietzner, A. Metzger, F. Leymann and K. Pohl. Variability Modeling to Support Customization and Deployment on Multi-Tenant-Aware Software as a Service Applications. In ICSE Workshop on Principles of Engineering Service Oriented Systems (PESOS), 2009.
- R. Mietzner, F. Leymann and M. P. Papazoglou. Defining Composite Configurable SaaS Application Packages Using SCA, Variability Descriptors and Multi-Tenancy Patterns. 3rd Intl. Conference on Internet and Web Applications and Services, pp 156-161, 2008.

- Nitu. Configurability in SaaS (software as a service) Applications. In Proceedings of the 2nd India Software Engineering Conference (ISEC), pp 19-26, 2009.
- K. Pohl, G. Bockle and F. Van Der Linden. Software Product Line Engineering: Foundations, Principles and Techniques. Springer-Verlag New York Inc, 2005.
- C. Weissman and S. Bobrowski. The Design of the Force.com Multi-Tenant Internet Application Development Platform. In SIGMOD, pp 889-896, 2009.
- D. Qi, A. Roychoudhury and Z. Liang. Test Generation to Expose Changes in Evolving Programs. In ASE, 2010.
- X. Ren, F. Shah, F. Tip, B. Ryder O. Chesley. Chianti: A Tool for Change Impact Analysis of Java Programs. In OOPSLA 2004, pp 432-448
- C. Treude, S. Berlik, S. Wenzel and U. Kelter. Difference Computation of Large Models. In ESEC/FSE 2007.
- T. Mens. A State-of-the-Art Survey on Software Merging. IEEE Transactions on Software Engineering, 28(5), 2002
- G. Brunet, M. Chechik, S. Esterbrook et al. A Manifesto for Model Merging. In Proceedings of GAMMA, May 2006
- M. Paloucci, T. Kawamura, T.R.Payne and K. Sycara. Semantic Matching of Web Service Capabilities. In Proceedings of IWSC, June 2002
- E. Stroulia and Y. Wang. Structural and Semantic Matching for Assessing Web Service Similarity. Int'l Journal of Cooperative Information Systems, 14(4), pp 407-438, 2005.
- Worldwide Software as a Service 2010-2014 Forecast: Software Will Never Be The Same. IDC Report Doc#223628, June 2010.
- Forecast: Public Cloud Services, Worldwide and Regions, Industry Sectors, 2009-2014. Gartner, June 2010.
- J.M.Kaplan. How SaaS is Overcoming Common Customer Objections. Cutter Consortium: Sourcing and Vendor Relationships, Advisory Service, Executive Update 8(9), 2008.