# Decoupling variability management in multi-tenant SaaS applications

Ali Ghaddar, Dalila Tamzalit, Ali Assaf

# Decoupling variability management in multi-tenant SaaS applications

Ali Ghaddar
*BITASOFT, University of Nantes*
*France*
*Email: ali.ghaddar@bitasoft.com*

Dalila Tamzalit
*University of Nantes, LINA*
*France*
*Email: Dalila.Tamzalit@univ-nantes.fr*

Ali Assaf
*BITASOFT*
*France*
*Email: ali.assaf@bitasoft.com*

***Abstract*—Variability represents an important challenge in multi-tenant SaaS applications. In fact, even if multi-tenancy realizes SaaS providers dream of having a single maintained software instance serving multiple customers (tenants) for common functionality, variations in tenants needs and their specific requirements at many places of the application bring providers back to the real world. They face an additional design concern: supporting application variability on a per-tenant basis. In this paper, we focus on such variability concern and try to reduce its complexity by decoupling its management through different application layers. We rely on a two-steps decoupling approach: the first step consists of representing application variations as an explicit variability model while the second step consists of choosing the must appropriate application layer(s) to manage each variation. Our approach is illustrated by relying on a case study from the food industry.**

***Keywords*-SaaS, multi-tenant, variability.**

## I. Introduction

The stability of the web and the reliable internet access has enabled the development of complex and robust software applications entirely on the web, generally proposed as services to potential customers. This approach enhanced the appearance of the Software-as-a-Service (SaaS) model. SaaS can be defined as: "Software deployed as a hosted service and accessed over the Internet" [7]. In fact, SaaS is a new software delivery model that permits its customers to leverage an on-demand software functionality through an Internet access, without the burden of deploying and managing the software themselves. Such new software delivery model has also an extension and evolution ability to support a cloud computing approach, by outsourcing to a third party providers an important part of, if not all, the infrastructure and platform resources required to run the SaaS application, following the Infrastructure-as-a-Service (IaaS) and the Platform-as-a-service (PaaS) delivery models [16]. SaaS and its associated principle of outsourcing by the underlying delivery models contrast from the traditional application server provider (ASP) model, where all costumers are hosted on the same provider data center, a costly approach for software providers especially in terms of infrastructure update and maintenance. However, from a SaaS provider perspective, the real benefits of SaaS begins when it is possible to host multiple customers on the same application instance, without the need of a dedicated application to be deployed and separately maintained for each customer. This approach is called *multi-tenant*, where tenant means a customer organizational structure, grouping certain number of users [4]. Multi-tenancy is more cost effective and easy to administrate, because the SaaS provider handles, updates and runs only a single instance of the software, eliminating the burden of handling multiple and different software releases. However, multi-tenancy compromise the software development and make it more complex, because the additional important concern of software variability needs to be correctly managed, thus each tenant feels like he is using a dedicated application. In this paper, we focus on such variability concern and try to reduce its added complexity by decoupling its management through different application layers. We will detail our approach and its context in the following.

The remainder of the paper is structured as follows. Section II presents the background of this research work as well as the problem that we address. Section III introduces our adopted variability management concept. Section IV shows a case study from the food industry, as our motivating example for decoupling variability management. Section V discusses variability management decisions. Section VI presents related works and Section VII concludes the paper.

## II. Background and Problem

Multi-tenancy makes the software development more complex. In fact, the software must support variability in its functional and non-functional behavior, in order to meet the specific requirements of each tenant, without affecting the other hosted tenants. Such development complexity could be amplified by the fact that multi-tenancy is a new principle in web applications development where many developers don't have enough skills or experiences to appropriately handle its variability requirements. SaaS providers are thus worry about the inherent costs of adopting a multi-tenant solution, which entails a re-education process for developers as well as investments in scientific researches in order to find an efficient and easy variability management solutions. A practical approach that we consider it may provide a good starting point for SaaS providers in this direction, would be exploiting the capabilities of the software architecture

supporting the SaaS application in order to face variability issue. In this paper, the software architecture being used for building our multi-tenant SaaS application is the service oriented one (SOA) and especially its web-services integration technology support, as a well suitable and flexible construction model for SaaS applications [17]. In fact, the benefits from using SOA in a SaaS context still an open issue, but usually, the two models are used interchangeably. SOA helps realizing SaaS applications quickly, through services reuse, and respectively for services reuse, decreasing the applications time to market and leveraging the economy of scale. However, when a multi-tenant SaaS application is built by a set of composed, orchestrated and interconnected services following the SOA principles, the variability management and implementation must be handled through the SOA's specifications and specificities. A SOA-based application involves multiple layers (see figure 3, lower part) for aims of separation of concerns and system flexibility, and hence, in such multi-layered architecture, the variability concern may impact and cut across the different layers. For this reason, the researches from the SOA community on the topic of variability has mainly focused on proposing different run-time adaptation and variability management mechanisms for each layer separately. The approaches proposed seems to be very efficient and their domains applicability are wide and divers. These approaches are divided under the three main SOA layers:

- The business-process layer (services orchestration and composition): The approaches proposed [15], [6], [8] try mainly to extend the business-process execution language (BPEL) [12] in order to support its run-time adaptation, especially that BPEL is recognized as a standard in the web services world.
- The service layer (functional layer represented as a set of invoked services): The approaches proposed [10], [22], [13] provide different adaptation mechanisms and patterns, making these individual (atomic) services more reusable in different run-time context.
- The interface layer (intermediate layer between business-processes and services): The approaches proposed [20], [14] try mainly to identify the conflict problems and the services interfaces mismatch at their input and output types, which is generally produced after a dynamic services replacement at the business-process layer.

Beyond SOA layers, and considering the different tenants preferences on their data management and storage, the data access layer may also get impacted by the variability concern. For this reason, various works [21], [1] have been provided to introduce tenant-specific data extensions as well as isolating tenants data that are hosted on the same database instance, while supporting different data access policies, storage and indexation.

However, these variability approaches and techniques are very different as well as the circumstances in which they could be applied. Having many solutions to address variability, each one tailored to a specific problem and at a specific layer, makes the decision on which layer(s) managing a particular variation a non-trivial and risky task. In addition, variability is generally thought on a given layer separately from the others, causing an unconscious isolation of its management, because in certain situations, some particular variations may requires two or more layers adaptation simultaneously. This issue will be detailed in the variability management decisions of our case study(See section V).

Another issue complicates variability management: according to our experience in multi-tenant development, SaaS providers usually receive variations in tenants requirements in an informal way and they are generally described following tenants experience and their domain terminology, which are very different from SaaS developers understanding and from their reference architecture's concepts. Thus, a significant effort must be done by developers to model and to transform these informal variations to an executable and maintainable code and, following our idea, it will take an effort to choose the most appropriate architecture layer(s) to implement and manage each variation.

To overcome these issues, we propose in this paper to model the application variations by relying on existing variability modeling techniques from the software product line engineering [2], [18]. Then, we propose to early think about variability in terms of the different layers of the application and try to decouple its management across these layers. We believe that such approach can reduce the long-term maintenance implications and the required development effort to integrate variability in the system.

## III. A CONCEPT FOR MULTI-TENANT VARIABILITY MANAGEMENT

A multi-tenant application has to be well designed and developed in order to support the variations in the tenants requirements. If not, tenants will loose the feeling of using a dedicated application and can doubt of the security of the separation between tenants and can thus decide to move on to another provider. In this context, the SaaS providers are certainly wondering how to offer such variability and to what extent it should be offered, in order to maintain the application ability to serve more and more tenants. One can say that offering a fixed set of options from which the tenants can select will always covers their different requirements, while some others may argue that a multi-tenant application must be highly flexible and totally configured via meta-data in all its services and functions. In fact, each approach has its own implications, while between these two approaches, their is many levels of variability that can be traversed. A SaaS provider who is looking to offer certain level of variability in

his multi-tenant application, he must be aware of his level of variability offer, in terms of its complexity impact on the application development and its efficiency to capture the maximum number of tenants. From our perspectives, we find that the balance between the variability offering and managing has to be well maintained, and we think that in order to justify the use of a shared application by different tenants, the commonality among their needs must be high enough to be at the same level of importance as variability. Thus, instead of adopting a highly flexible and expensive design which focus on variability as a primary design concern, we have adopted an intermediate design solution that exploits the tenants commonalities and suitably managing the application variability each time needed. At the implementation level, this intermediate approach is realized by letting the SaaS developers start implementing the tenants commonalities, without making any implementation decision at variation places. Meanwhile, another team of developers which we have classified as a variability experts team, will be providing variants for these variation places and implementing a run-time variability management mechanism. Such management mechanism is responsible of the dynamic bind of variants to their respective places of variations in the application. The researches from the software product engineering (SPLE) on the topic of variability can be well applied in this situation, since SPLE defines *variation points* to which different variants may be linked. In fact, we strongly believe that the multi-tenant variability modeling and managing could be handled through these two SPLE concepts, but some technical improvements at the implementation level must be done before, in order to meet the run-time variability challenge of the multi-tenant model, while SPLE have focused more on a compile-time variability support [23]. Each variant in a multi-tenant application can represent a tenant-specific need, while some other variants may be common for many tenants. When a new tenant onboard the system, he can select an existing variant, or may demand the creation of a new one if no existing variant matches his specificities. The information about the tenants variants can be stored in external configuration files. At run-time, the calling tenant is identified, and the concerned variation points from the call must be bound to the current tenant related variants, in order to realize an adapted application behavior on a per-tenant basis (see figure 1).

Technically, managing multi-tenant variability through such intermediate approach would not be possible without taking into account the variability concern at the early stages of the application design, by explicitly identifying variation points as well as their respective variants, while preparing their run-time binding and management mechanism. In our multi-layered architecture, a variability management mechanism has to be made available for each application layer separately due to the differences between these layers technologies and concepts. This will enable us to early decouple the application variability management through the different layers and thereby, each variation point can be managed at its best suited application layer(s). These early steps to make has to prepare the ground for variability and fall under the concept of *early variability* that we propose, to derive our way of thinking in multi-tenant applications design and development.

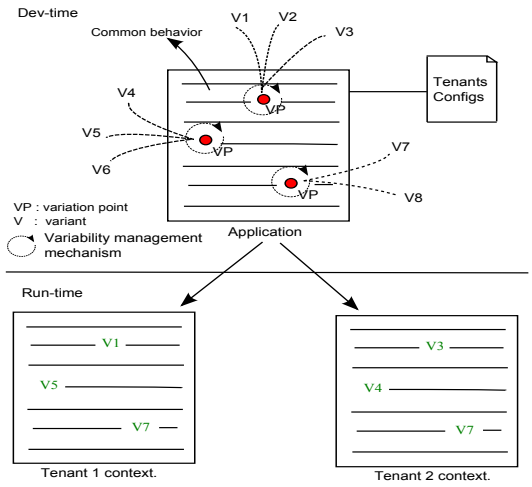


Figure 1. Variation points and variants at both development and run-time

## IV. Case Study: A Food Industry Application

In this paper, we choose a food-industry application (FIA for short) that we have developed, as our case study. FIA is a SOA-based multi-tenant application. It allows its food industry tenants looking for foods production and quality improvement to predict the future expenses and benefits from their potential recipes, before executing a real and expensive fabrication process. FIA evaluates, by relying on an internally developed simulation service, the recipes fabrication time and cost, as well as their quality characteristics such as nutritional value, taste and smell. FIA has also an integrated shipping service for ingredients, from the warehouse of the ingredients supplier (partner of FIA), to the tenants manufactures. Figure 2 shows the FIA application business-process.

Three main actors exist in the process: tenant user, service provider and ingredients shipper. In the following we will explain the role of each one.

1) **Tenant user:** After accessing the application, the user can manage his existing recipes or decide to create a new one. In this last case, user has to provide the ingredients composing the new recipe as well as their respective percentages in it (user can specify his ingredients choice by using their universal codes from ”CIQUAL or USDA” ingredients bases). The

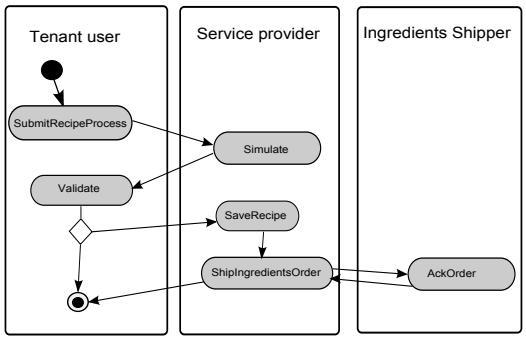


Figure 2. FIA Business process

user must also describe the recipe cooking process, by sending a pre-negotiated XML structure, or he can rely on a FIA integrated cooking process drawing tool. When finishing, the user must click on simulate button and wait for results.

2) **Service provider:** After receiving the ingredients and the cooking process of the recipe, the ingredients prices and quality informations are retrieved from the database, they have been originally stored by the ingredients supplier partner with a dedicated user interface, and they are always under updates as prices and quality informations change. Those ingredients informations as well as the recipe cooking process described by the user, allow the simulation service to evaluate the final recipe cost, fabrication time and quality. However, when the simulation ends, the application sends back a simulation report to the user, containing the recipe properties values, which the tenant has interest to keep a good balancing between them, for financial, economic and market needs. Once user validate the simulation, the recipe details are saved in the database and an ingredients shipping order is sent.
3) **Ingredients shipper:** he is a third-party actor, offering a shipping service from the ingredients warehouse to the tenant manufacture. When the tenant receives the ingredients, he can execute a real recipe fabrication process, being sure that the recipe will have its predicted properties values.

### A. FIA variations

In order to *early* detect possible variations in the application, we have presented FIA business process to some selected potential tenants. Five variations have been detected: 1) some tenants are interested by the software but they have their own ingredients suppliers. 2) Some other tenants asked for another ingredients shipper because the existing one is costly. Taking into account this shipping variation, we decided to add different shippers to the system, each one offers different shipping time and cost. However, having an existing ingredients supplier for some tenants excludes the need of the FIA shippers, because usually all ingredients suppliers have their own shippers. 3) Another variation has been detected in the way that the simulation service must behave. In fact, for an accurate simulation, different factors related to the tenants manufactures has to be considered, impacting the recipe simulation results (fabrication time, cost and quality). Factors like manufactures energy consumption, waste of ingredients on production chain, number of workers, speed of execution, machines quality effect on ingredients, and many unknown other factors, that they will begin to appear when new tenants on-board the system. 4) We have also found a variation in the way that the recipes details must be stored and retrieved from the database. As FIA is a multi-tenant application, tenants recipes are stored on the same database instance. SaaS provider must thus ensure recipes isolation. However, some tenants may have some specific security rules that prevent storing their recipes details in a shared database. Thus, FIA offers these tenants the possibility to store their recipes details in a dedicated databases. 5) Finally, we have found that some tenants belong to the same multi-national group, and they would like to share their recipes details in order to leverage each others information and expertises. However, when choosing a separated database, sharing recipes details will no longer be possible.

### B. Modeling FIA variations

The main reason for supporting variability in FIA is to enhance its availability and adaptation to different tenants financial, business and security needs. However, having many variations as well as dependencies between them, leads to a complexity in variability specifications, motivating the need of a variability model that helps us to trace application variations and takes the right management decisions. In Figure 3 upper part, we model FIA variations by relying on the Orthogonal Variability Model (OVM) introduced in [18]. OVM consists of the two main modeling concepts: *variation point* (VP) and *variant*. Variants in OVM can be either *mandatory* or *optional*. A mandatory variant must be selected on a given variation point, while many optional variants can be selected from a given group of variants, their minimum and maximum number (min, max) specify the range for the permissible numbers of variants to be selected from the group. OVM also defines constraints between variants, between VPs and between variants and VPs: An *exclude* constraint specifies a mutual exclusion; e.g., if variant1 at VP1 excludes variant2 at VP2, the variant2 can not be used at VP2 if variant1 is used at VP1. A *requires* constraint specifies an implication; i.e. if a variant is used, another variant has to be used as well.

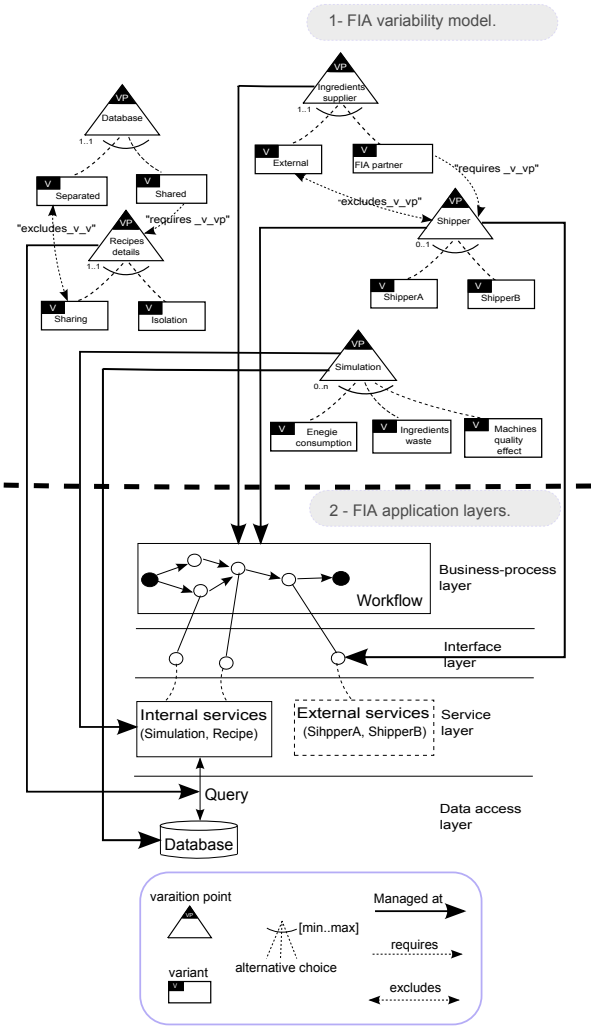


Figure 3. Modeling variability and decoupling its management through different application layers

## V. Decoupling Variability Management

After modeling variability, the software architect and the variability experts team must take the right decision about where managing these variations in the different application layers. In the remainder of this section, we describe how FIA variations are resolved and we show how the complexity of variability management can be reduced if these variations are managed at their most appropriate layers.

### A. Suppliers and Shippers Variations

These two variation points are very close in terms of their context and their management decisions, and they will be explained together. We will start by the supplier variation. According to this last variation specification, some tenants need that FIA interact with their own ingredients suppliers instead of local one (FIA partner). Such interaction has a main goal to retrieve the ingredients informations (price and quality) that compose the recipe, thus, they could be used in the evaluation of the recipe properties. In the FIA business process, the behavior responsible of retrieving the ingredients informations was thought to be a part of the simulation service design. Managing this variation inside the simulation service will obligates the importation of all external suppliers data in our database, while using some conditional code or static inheritance in the service design, in order to switch between different suppliers according to the calling tenant requirements. With the arrival of new tenants, this conditional code must be modified and new data must be imported in order to add new external suppliers into the system, which is very costly to manage and to maintain, while tenants choices of suppliers can no more be modified at run-time. Thus, we have decided to benefit from the SOA capabilities and especially its web-services integration technology, by implementing web-services on top of the different information systems of external suppliers, as a pre-condition for those suppliers to be accepted and added to our system. This will eliminates the need of data importation. In addition, according to such new web-based nature of all external suppliers services, we can switch between these services at the BPEL, by dynamically changing the service reference (partner link), which is easy to maintain and enables the run-time modifications. However, since there is no existence of a supplier service in our business process design, we *early* manipulate the simulation service, by separating the behavior responsible of retrieving ingredients informations and modularize it in a separated supplier service, partner of FIA, thus switching between partner and external suppliers services variants could be possible. We can also proceed in the same way for the shipper variation point, by switching between the two shipping services according to tenants requirements or ignoring their call for tenants having external suppliers. This type of adaptation is widely accepted and commonly used [15] in the web-services composition and orchestration.

### B. Simulation Variation

In order to ensure an accurate simulation for each tenant, several variants should be provided to consider the different machines and manufacture properties of the tenants, such as their manufactures energy consumption, the ingredients waste on their production chains, their machines quality effect on ingredients, etc. Each variant introduces its modification to the simulation results. A rapid solution to manage this variation would be creating a simulation service per-tenant, and adapting the services composition at the business-process layer, by selecting the appropriate version of the simulation service that corresponds to the calling tenant. This type of adaptation has a long-terms maintenance implications on the system, because it will became very complex to maintain the growing number of services as new

tenants on-board the system, in contrast to the suppliers services, where a particular supplier may be common for many tenants. After analysing the context of this simulation variation, we have found that its variants can be treated as separated concerns from the simulation common behavior. Managing this variation will be improved if we factorize the common behavior of this service into one core service, while implementing one adaptation variant of this core service per-tenant. In our case, the common behavior of the service is cooking the recipe following the described process, which is independent from the above variants. Managing such variation could be done using a dynamic aspect-oriented programming (AOP) [19] mechanism, by implementing each variant as an independent aspect. Those aspects will be dynamically weaved into the core service, according to the calling tenant requirements (aspects will not be weaved if the variants that they represent are not selected by the calling tenant), before returning the service results. Each aspect has to retrieve from the database, one of the values of the tenant manufacture and machines properties, which they are stored as a tenant-specific data extensions, since they differs from one tenant to another. Each value will be calculated within its dedicated aspect in order to evaluate and to apply its impact on the simulation results, following some predefined rules and equations.

### C. Recipes Details Variation

This variation consists on allowing tenants to share their recipes details with other tenants in the same group, or keeping their recipes details isolated. Managing this variation by adapting the services composition at the business-process layer requires having two recipes services variants, one ensuring isolation and second allowing sharing. This way of management will be complex to maintain, since the same variation nature could be met in the future within other services. In fact, the data sharing concern is probably democratized for different application services. For example, in the *knowledge-base* service that we plan to develop, tenants would ask to share their research articles. After well analysing the context of this variation , we have found that its management must not affect on the internal design of the recipe service, neither all future services that may interact with the database to make some create, read, update or delete (*CRUD*) operations. The developers have to develop and use such kind of services without thinking in multi-tenancy. Thus, we have decided to manage this variation at the data access layer, by letting developers write the *CRUD* operations queries for the service in a single-tenant way, while adjusting those queries for data isolation or sharing. Adjusting queries is done by relying on some *predefined* query extensions, which are developed once for all concerned services. This will be transparent to developers and therefore easy to maintain. However, enabling this type of variability management requires adding two new columns to each concerned table in the shared database: `TenantID` and `GroupID`. To retrieve and store records in the shared tables, a query adjuster mechanism is used to apply query extensions. For example, to get all recipes records, the *base* query written by a developer is as follows:

```
SELECT * FROM RECIPE
```

The first extension of this query ensuring recipes isolation is:

```
SELECT * FROM RECIPE
WHERE TenantID='123'
```

The second extension allowing recipes sharing is:

```
SELECT * FROM RECIPE
WHERE(TenantID='123' OR GroupID='3')
```

In addition, another reason to *early* think about separating *base* query is the existence of the database variation. In fact, some tenants may ask for a dedicated database variant. Supporting this variation will be complex if the developers directly write the whole query, while the separation of base queries allows a native support of such variant without having to rewrite all queries for storing and retrieving data, when installing dedicated databases.

## VI. Related Work

### A. SOA variability management

Several authors studied the variability concern in the context of SOA systems. In [5] authors identify four types of variability which may occur on SOA. In[23] authors present a framework and related tool suite for modeling and managing the variability of Web service-based systems. They have extended the COVAMOF framework for the variability management of software product families. In [11] authors describe an approach to handle variability in Web services, while [15] focus on variability in business-process by proposing a VxBPEL language. However, all these approaches focus on variability in SOA systems, but they do not address variability in multi-tenant SaaS context.

### B. SaaS and Multi-Tenancy

In [9] authors provide a catalog for customization techniques that can guide developers when dealing with multi-tenancy, they have identified two types of customization: Model View Controller (MVC) customization and system customization. In [3] the authors propose some architectural choices to make when building multi-tenant applications, while in [4] authors discuss their experiences with re-engineering an existing industrial single-tenant application into a multi-tenant one. In [17], authors propose a variability modeling technique for SOA-based multi-tenant applications, they differentiate between internal variability only visible to the developers, and external variability that is communicated to the

tenants of the application. However, they do not address the variability management according to the different application layers, which we have treated in this paper.

## VII. Conclusion and Future Work

In this work, we have shown the importance and the complexity of supporting variability at the early stages in multi-tenant applications. We have also motivating the need of supporting the variability at different application layers. We have started from a variability model of a concrete case study and we have shown different techniques for managing variability at each system layer.

Our future work consists of extracting variations properties that have influenced our decision about the layer of managing variability. Once these properties are identified, they can be evaluated at each variation. This can enormously help architects and developers to take the right decision and to accelerate this heavy task.

## References


[1] 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 2008 ACM SIGMOD international conference on Management of data*, pages 1195–1206. ACM, 2008.

[2] J. Bayer, S. Gerard, O. Haugen, J. Mansell, B. Moller-Pedersen, J. Oldevik, P. Tessier, J.P. Thibault, and T. Widen. Consolidated product line variability modeling. 2006.

[3] C.P. Bezemer and A. Zaidman. Multi-tenant saas applications: maintenance dream or nightmare? In *Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE)*, pages 88–92. ACM, 2010.

[4] C.P. Bezemer, A. Zaidman, B. Platzbeecker, T. Hurkmans, and A. t Hart. Enabling multi-tenancy: An industrial experience report. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pages 1–8. IEEE, 2010.

[5] S.H. Chang and S.D. Kim. A variability modeling method for adaptable services in service-oriented computing. 2007.

[6] A. Charfi and M. Mezini. Ao4bpel: An aspect-oriented extension to bpel. *World Wide Web*, 10(3):309–344, 2007.

[7] F. Chong and G. Carraro. Architecture strategies for catching the long tail. *MSDN Library, Microsoft Corporation*, pages 9–10, 2006.

[8] O. Ezenwoye and S.M. Sadjadi. Trap/bpel: A framework for dynamic adaptation of composite services. In *Proceedings of the International Conference on Web Information Systems and Technologies (WEBIST 2007)*. Citeseer, 2007.

[9] S. Jansen, G.J. Houben, and S. Brinkkemper. Customization realization in multi-tenant web applications: case studies from the library sector. *Web Engineering*, pages 445–459, 2010.

[10] H. Jegadeesan and S. Balasubramaniam. A method to support variability of enterprise services on the cloud. In *Cloud Computing, 2009. CLOUD'09. IEEE International Conference on*, pages 117–124, 2009.

[11] J. Jiang, A. Ruokonen, and T. Systa. Pattern-based variability management in web service development. In *Web Services, 2005. ECOWS 2005. Third IEEE European Conference on*, pages 12–pp. IEEE, 2005.

[12] D. Jordan, J. Evdemon, A. Alves, A. Arkin, S. Askary, C. Barreto, B. Bloch, F. Curbera, M. Ford, Y. Goland, et al. Web services business process execution language version 2.0. *OASIS Standard*, 11, 2007.

[13] Y. Kim and K.G. Doh. Adaptable web services modeling using variability analysis. In *Convergence and Hybrid Information Technology, 2008. ICCIT'08. Third International Conference on*, volume 1, pages 700–705. IEEE, 2008.

[14] W. Kongdenfha, R. Saint-Paul, B. Benatallah, and F. Casati. An aspect-oriented framework for service adaptation. *Service-Oriented Computing–ICSOC 2006*, pages 15–26, 2006.

[15] M. Koning, C. Sun, M. Sinnema, and P. Avgeriou. Vxbpel: Supporting variability for web services in bpel. *Information and Software Technology*, 51(2):258–269, 2009.

[16] F. Leymann and D. Fritsch. Cloud computing: The next revolution in it. *Proceedings of the 52th Photogrammetric Week*, pages 3–12, 2009.

[17] 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 2009 ICSE Workshop on Principles of Engineering Service Oriented Systems*, pages 18–25. IEEE Computer Society, 2009.

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

[19] A. Popovici, T. Gross, and G. Alonso. Dynamic weaving for aspect-oriented programming. In *Proceedings of the 1st international conference on Aspect-oriented software development*, pages 141–147. ACM, 2002.

[20] Y. Sam, O. Boucelma, and M.S. Hacid. Web services customization: a composition-based approach. In *Proceedings of the 6th international conference on Web engineering*, pages 25–31. ACM, 2006.

[21] O. Schiller, B. Schiller, A. Brodt, and B. Mitschang. Native support of multi-tenancy in rdbms for software as a service. In *Proceedings of the 14th International Conference on Extending Database Technology*, pages 117–128. ACM, 2011.

[22] M. Stollberg and M. Muth. Service customization by variability modeling. In *Service-Oriented Computing. ICSOC/ServiceWave 2009 Workshops*, pages 425–434. Springer, 2010.

[23] C. Sun, R. Rossing, M. Sinnema, P. Bulanov, and M. Aiello. Modeling and managing the variability of web service-based systems. *Journal of Systems and Software*, 83(3):502–516, 2010.