The Journal of Systems and Software 91 (2014) 48–62

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Efficient customization of multi-tenant Software-as-a-Service applications with service lines

Stefan Walraven[a,*], Dimitri Van Landuyt[a], Eddy Truyen[a], Koen Handekyn[b], Wouter Joosen[a]

[a] iMinds-DistriNet, KU Leuven, Celestijnenlaan 200A, 3001 Leuven, Belgium
[b] UnifiedPost, Avenue Reine Astrid 92A, 1310 La Hulpe, Belgium

ARTICLE INFO

Article history:
Received 15 November 2012
Received in revised form 9 January 2014
Accepted 13 January 2014
Available online 22 January 2014

Keywords:
Multi-tenancy
SaaS
Variability

ABSTRACT

Application-level multi-tenancy is an architectural approach for Software-as-a-Service (SaaS) applications which enables high operational cost efficiency by sharing one application instance among multiple customer organizations (the so-called tenants). However, the focus on increased resource sharing typically results in a one-size-fits-all approach. In principle, the shared application instance satisfies only the requirements common to all tenants, without supporting potentially different and varying requirements of these tenants. As a consequence, multi-tenant SaaS applications are inherently limited in terms of flexibility and variability.

This paper presents an integrated service engineering method, called *service line engineering*, that supports co-existing tenant-specific configurations and that facilitates the development and management of customizable, multi-tenant SaaS applications, without compromising scalability. Specifically, the method spans the design, implementation, configuration, composition, operations and maintenance of a SaaS application that bundles all variations that are based on a common core.

We validate this work by illustrating the benefits of our method in the development of a real-world SaaS offering for document processing. We explicitly show that the effort to configure and compose an application variant for each individual tenant is significantly reduced, though at the expense of a higher initial development effort.

1. Introduction

Software as a Service (SaaS) has become a common software delivery model. It is a form of cloud computing that involves offering software services in an on-line and on-demand fashion (with the Internet as the main delivery mechanism). One of the key enablers in cloud computing to achieve economies of scale is *multi-tenancy* (Chong and Carraro, 2006; Guo et al., 2007): the sharing of resources among a large group of customer organizations, called *tenants*. This architectural concept can be applied at various levels of the software stack: at the infrastructure level (i.e. virtualization), at the OS and middleware level, and even at the application level. Each approach makes a different trade-off between (i) maximizing scalability and operational cost benefits (including hardware and software resource usage as well as maintenance effort), and (ii) maximizing flexibility to meet the potentially different and varying tenant-specific requirements (Walraven et al., 2011).

This paper focuses on application-level multi-tenancy. Application-level multi-tenancy achieves the highest degree of resource sharing between tenants (Chong and Carraro, 2006; Walraven et al., 2011). End users from different tenants are simultaneously served by a single application instance on top of shared infrastructure. However, when compared to infrastructure-level and middleware-level multi-tenancy, the inherent limitations in variability form a crucial disadvantage. The high degree of resource sharing typically results in a one-size-fits-all approach: the multi-tenant application only satisfies the requirements that are common to all tenants. Support for different and varying requirements of the different tenants is lacking.

To shorten the time-to-market, the initial development and release cycles of a SaaS application are typically focused on the

* Corresponding author. Tel.: +32 16327640.
E-mail addresses: stefan.walraven@cs.kuleuven.be (S. Walraven), dimitri.vanlanduyt@cs.kuleuven.be (D. Van Landuyt), eddy.truyen@cs.kuleuven.be (E. Truyen), koen.handekyn@unifiedpost.com (K. Handekyn), wouter.joosen@cs.kuleuven.be (W. Joosen).

0164-1212/$ – see front matter © 2014 Elsevier Inc. All rights reserved.
http://dx.doi.org/10.1016/j.jss.2014.01.021

needs of the first customer organizations (tenants). As the SaaS offering becomes more successful, an increasing amount of variations (ranging from minimal to substantial) is implemented to service new tenants and an increasing amount of tenant-specific configurations therefore have to co-exist at run time. This easily leads to an explosion of relatively small variations in the implementation as well as in the different configurations. This real-world scenario is experienced in many specific business cases; we have identified a lack in *methodical support for the development and customization of multi-tenant applications*. This lack of support can be characterized by two essential challenges:

First, SaaS providers need to be able to *manage and reuse the different configurations and software variations in an efficient way*, without compromising scalability; e.g. by avoiding additional overhead when provisioning new tenants.

Secondly, part of realizing the scalability benefits of SaaS is achieved by *self-service*: shifting some of the configuration efforts to the tenant side, e.g. by allowing the tenant to manage his tenant-specific requirements and by automating the run-time configuration process. Therefore, tenants require *additional support to manage the configuration in a tenant-driven customization approach*.

In the state of the art, some work has been performed to combine the benefits of software product line engineering (SPLE) (Clements and Northrop, 2001; Pohl et al., 2005) with those of multi-tenancy to facilitate the customization of SaaS applications tailored to the tenant-specific needs (Mietzner and Leymann, 2008; Mietzner et al., 2009; Walraven et al., 2011; Schroeter et al., 2012a). We define a *service line* as a specific concept that leverages on the notion of software product lines by offering a shared application instance that still is dynamically customizable to different tenant-specific requirements. However, none of the current service engineering approaches offer a complete customization process for multi-tenant applications. Moreover, customization is often limited to specific technologies or to specific types of applications.

Our solution is a feature-oriented method that is highly integrated, in the sense that the feature-level variability that is introduced in the early development stages is consistently and explicitly supported in each of the subsequent development stages, also in the run-time environment. This is a key difference with respect to traditional SPLE. Instead of delivering a dedicated, separate application product for each tenant (cf. the application engineering phase in SPLE), the entire service line (including all variations) is instantiated and deployed only once and simultaneously shared by all tenants. Specific software variants are activated at run time within one single SaaS application instance.

The main contribution of this paper is an integrated service line engineering method that focuses on addressing variability up front without compromising the scalability of SaaS applications. This method starts with the initial development phases (requirements engineering, architectural design and implementation) of the service line, but also focuses on the deployment, run-time configuration and composition, the operations and maintenance. The method is generic in the sense that each stage is open for existing work in the state of the art to be leveraged upon, yet it imposes some specific constraints (e.g. composability and traceability of features) and requires some enablers (e.g. multi-tenancy, feature-level versioning, tenant-level configuration interfaces) for service line variability.

We have validated this method in the development of an industry-level SaaS application in the domain of online B2B document processing. To this end, we closely collaborated with an industrial SaaS provider in the context of a collaborative research project (CUSTOMSS, 2011). Our evaluation focuses on illustrating (i) the efficiency benefits with respect to addressing the management complexity of many co-existing tenant-specific configurations, and (ii) the trade-off between the *early* effort required to design and implement the initial service line, and the *late* effort required to configure and compose application variants, to provision new tenants as well as to update and maintain the service line as a whole.

The structure of this paper is as follows. Section 2 motivates this work and elaborates on the problem statement. Section 3 articulates the concept of a service line and describes the service line engineering method. In Section 4, this method is applied on a SaaS application in the domain of document processing. We evaluate and discuss our work in terms of the efficiency benefits associated with service lines in this specific SaaS application in Section 5. In Section 6, related work is discussed and Section 7 concludes the paper.

2. Problem elaboration

The motivation for this paper is based on our extensive insight into the current state of practice of a number of industrial SaaS providers (which was obtained in the context of several applied research projects, e.g. CUSTOMSS, 2011). In this section, we first present our characterization of this current state of practice of developing, operating, and maintaining multi-tenant SaaS applications. Then, we list the main challenges that are addressed in this paper.

2.1. State of the practice

We present a number of development and management activities that occur in the lifecycle of a multi-tenant SaaS application. Specifically, we focus on the customization and management challenges that SaaS providers face to efficiently offer and manage their offerings.

The following stakeholders are involved in these scenarios. The *SaaS architect*, *SaaS developer* and *SaaS operator* are employees of the SaaS provider. In addition, each tenant should assign a *tenant administrator* role to the person responsible for managing the SaaS application on behalf of the tenant organization (e.g. for user and configuration management). In theory, this role can be assigned to an internal user of the tenant or to the SaaS provider, or even to value-added resellers who are a business channel between tenants and the SaaS provider. In practice, the tenant administrator is often an employee of the SaaS provider, as detailed technical knowledge of the SaaS application is required.

Scenario #1: Initial development of the SaaS application. This scenario entails the initial design and development of a multi-tenant SaaS application by the SaaS architect and the SaaS developers. The SaaS operator is responsible for deploying and managing the (running) implementation of this SaaS application.

To shorten the initial time-to-market, the first development cycles of a SaaS application are typically focused strongly on the needs of the first customer organizations (tenants). As a result, the SaaS application typically supports limited variability beyond the scope of the initial tenants.

Scenario #2: Provision a new tenant. In this scenario, a new tenant wants to use the SaaS application and customize it to its requirements. We assume that the SaaS application already covers the requirements of the new tenant, e.g. because these requirements are very similar to those of already provisioned tenants. (Scenario #4 covers the case where new requirements have to be supported.)

Based on his (technical) knowledge about the application, the tenant administrator has to manually translate the tenant's requirements into a software configuration for the multi-tenant SaaS application. Obviously when certain requirements of the new tenant are similar to those of other tenants, parts of the existing configuration can be reused. In practice, this is done in a weakly controlled and error-prone manner (e.g. by copy-pasting

configuration fragments). As a consequence, validation and testing of the newly-created configuration is of high importance. With an increasing number of tenants, this manual approach easily leads to an explosion of relatively small variations in the tenant-specific configurations.

Scenario #3: Update tenant-specific configuration. The requirements of a tenant can change over time. To satisfy these varying requirements, the tenant-specific configurations have to be updated.

Every time the requirements of a tenant change, the tenant administrator has to update the configuration to fit the new requirements and to ensure correct behaviour. Again, this is an error-prone manual task, and it might lead to divergence of configuration fragments that were initially equal (e.g. by copy-pasting), resulting into even less reuse and manageability of configurations.

Scenario #4: Support new requirements. In this scenario, some requirements of existing or new tenants are not yet covered by the current SaaS application. Therefore, the SaaS architect and developers have to add support for new requirements. This implies that new variations are introduced into the SaaS application.

Adding support for new requirements can have an impact on the existing workflows, services, components and interactions of the SaaS application. Such changes will ripple to the tenant-specific configurations, even to the configurations of tenants whose requirements did not change. The SaaS provider has no view on configurations that are affected, and the SaaS operator will have to verify the configurations of all tenants, while the tenant administrator will have to double-check specific configurations. This manual task obviously does not scale well with the large amount of tenants.

Scenario #5: Update and maintain SaaS application. The SaaS operator is also responsible for the maintenance of the SaaS application as well as the underpinning platform, and for keeping these up-to-date. For example, the current implementation might have to be upgraded to a new version of the platform or libraries, bugs have to be fixed, or performance improvements have to be introduced. It is a key advantage of SaaS applications that these updates can be performed more frequently than in an on-premise setting, i.e. maintenance happens continuously. On the one hand, such changes can be limited to the internals of a single software component, but on the other hand they can also affect the complete architecture of the SaaS application and even conflict with the requirements of some tenants. Similarly to the previous scenario, the SaaS operator and tenant administrator will have to verify the configurations and correct them if needed.

In the case that the updates conflict with the current preferences of some tenants, the tenant administrator cannot solve the issue by updating the configuration. A solution for this issue is to support both the older and the latest version of the changed software artifacts simultaneously, possibly for a limited time only (transition period). However, this implies that the SaaS operator should maintain different versions, and this easily leads to a rapidly growing number of relatively small variations in the implementation as well as in the different tenant configurations.

2.2. Challenges

Based on the above characterization of the current state of practice, and based on the state of the art (which is presented in Section 6), we have identified two key challenges to support tenant-specific customization in multi-tenant SaaS applications:

1. *Efficient development, management and reuse of software variations in a SaaS offering*. As illustrated above, variability in multi-tenant SaaS applications may have fundamentally different sources (tenant's requirements, evolution, maintenance, etc.), but is inherently present. Currently, software variations and tenant-specific requirements are introduced and managed in an ad-hoc, often manual and error-prone fashion, and we have shown above that this has some obvious disadvantages. For example, whenever a tenant's requirement changes, the development process has to be re-executed for this tenant, and the ensuing changes might cause ripple effects that affect potentially all tenants. Especially when the SaaS offering evolves and the amount of supported variations grows, the cost of variability (expressed in terms of development, maintenance, and configuration effort) grows accordingly.

   Because of the high impact of variability on the SaaS application, it cannot be realized as an afterthought in the development process. Essential qualities such as maintainability and evolvability of the SaaS application, as well as the reusability and modularity of specific software variants must be taken into account throughout the entire development life-cycle of the SaaS application (in accordance with design principles such as attribute-driven design (Bass et al., 2003)).
2. *Efficient, tenant-driven customization of the multi-tenant SaaS application*. In the current state of practice, the tenant administrator has to analyze the tenant's requirements, create technical configurations, validate, test and activate these configurations manually in the SaaS application. One of the main advantages of the SaaS paradigm however is related to the fact that some of the configuration complexity can be outsourced to the tenant himself (typically called *self-service* (Sun et al., 2008)). This is essential to allow the SaaS provider to benefit from cloud scalability benefits; i.e. by reducing the time to provision new tenants.

   Specifically, dedicated configuration interfaces must be offered to the tenant that (i) are sufficiently expressive to allow him to create valid configurations of the SaaS application that meet his requirements (i.e. expressed in terms of the variations supported by the SaaS provider), but (ii) that hide the technical inner workings of the SaaS application from the tenant (i.e. provide abstraction). In addition, some of the activities of the tenant customization process (generation of technical configurations, validation, run-time activation of configurations) have to be automated to achieve this level of self-service.

In this paper, we address these challenges by introducing an integrated service line engineering method which is presented in the next section.

3. Service line engineering: concepts and method

This section presents our *service line engineering (SLE) method* that supports tenant-level variability in multi-tenant SaaS applications without compromising the essential benefits of scale associated with cloud computing. We define a *service line* as a SaaS application that is built as a software product line consisting of customizable services that can be dynamically selected and configured based on the tenant-specific requirements, with the major difference that *one single instance is supporting the different application variants*.

The SLE method is feature-oriented and highly integrated, in the sense that the feature-level variability that is introduced in the early development stages is consistently and explicitly supported in each of the subsequent development stages, also in the run-time environment. Fig. 1 presents the individual development and management activities and their input and output artifacts of relevance to the presented method.

The initial investment is in Service Line Development and Deployment, which consists of the following activities:

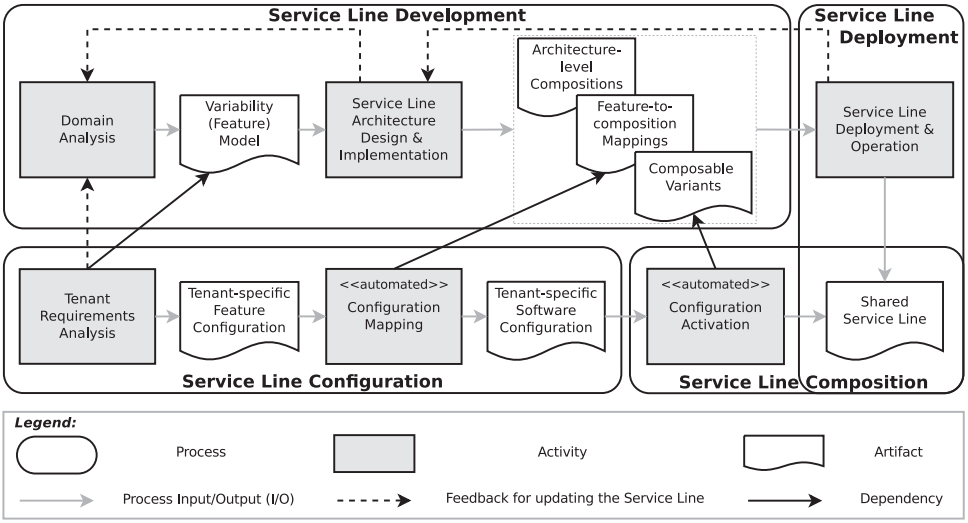

Fig. 1. Service line engineering method.

1. **Domain Analysis** is quite similar to SPLE, yet it introduces a feature-based variability model that *includes SLAs* and *supports versioning* of the feature model itself, both of which are key requirements for service lines.
2. During **Service Line Architecture Design & Implementation**, variability is made explicit at the level of the service line architecture by creating separate and explicit *variability-supporting views*. Abstraction is made of (i) the level of granularity of the service line compositions supported in the architecture and (ii) the underpinning SaaS middleware and dynamic composition technologies that will be used. Additionally, the traceability and composability of features and their corresponding software variants are taken into account as key architectural requirements for the service line. The implementation support is based on *feature-to-software-composition mapping specifications* that describe the mapping between feature models and the selected features, and specific components in the full architecture of the service line. These mappings are the basis for automation.
3. **Service Line Deployment** involves the instantiation of the service line on a powerful platform, so that the impact of technical developments to complement and extend PaaS, IaaS and middleware technology can be limited. We present some *key requirements and criteria* for these types of platforms, without going into the details of comparing a snapshot of some popular, contemporary technologies. After initial service line development and deployment, the focus shifts towards **Service Line Operation**. To avoid service disruption while conducting essential evolution and maintenance activities, the service line and the underpinning middleware should support non-trivial dynamic adaptations far beyond the classical upscaling or downscaling of resources typically cited in a SaaS context, e.g. updating the underpinning platform or supporting new features.

Subsequently, Service Line Configuration and Composition become relatively straightforward by leveraging on the investment of service line development and deployment.

1. **Tenant Requirements Analysis** leverages on the feature model (defined during domain analysis): a feature-oriented configuration interface is offered to the tenants, and verification of the selected set of features is done by validating the constraints of the feature model.
2. **Configuration Mapping** generates an application variant, in fact a tenant-specific configuration that extends the service line implementation, on the basis of the mapping specifications that have been defined in the service line architecture.
3. Finally, **Configuration Activation** occurs at run time. When requests arrive that can be attributed to a specific tenant, the running service line is dynamically adapted to match the tenant-specific configuration. This becomes feasible because of (i) the composability and traceability of variants, (ii) the modularity of tenant-specific configurations that co-exist in the service line, and (iii) the capabilities of the underpinning platform.

As with many software engineering approaches that deal with evolving requirements (Nuseibeh, 2001; Etien and Salinesi, 2005), it is an essential property of the presented SLE method that it is iterative. This is represented by the multiple feedback loops in Fig. 1. Many of these feedback loops are driven by the scenarios presented in Section 2.1, for example new requirements that are discovered during the tenant requirements analysis often lead to a re-iteration over the previous activities. These loops and the activities they pertain to, are discussed below.

3.1. Domain analysis

Next to regular requirements elicitation and analysis techniques (obtaining functional and non-functional requirements), engineering a service line starts with the activity of domain analysis to obtain the essential characteristics of SaaS applications in a particular domain (e.g. document processing). Variability analysis is a part of domain analysis that focuses on eliciting the typical commonalities and variabilities within the service line. We focus on tenant-level variability in this activity, i.e. differences in tenant requirements (sometimes also called *external variability* (Pohl et al., 2005)). A key output of this activity is the *variability model* that represents the domain-specific commonalities and variabilities in a service line.

In this paper (and in alignment with traditional SPLE), we apply a feature-based approach (Kang et al., 1990, 1998) to model variability, and therefore the relevant output variability models we discuss in this paper are feature models. The state of the art covers several approaches for feature modeling and feature models can be represented in different ways, from text-based (e.g. FDL by van Deursen and Klint, 2002 and TVL by Classen et al., 2011) to graphical in the form of feature diagrams (e.g. FeatureIDE by Kastner et al., 2009 and S.P.L.O.T. by Mendonca et al., 2009). Furthermore, CVL (Haugen et al., 2012) is a language for variability modeling that is submitted for standardization.

This activity does not differ substantially from domain analysis in traditional SPLE approaches. To build the initial feature model, requirements are gathered from (internal) domain experts, as well as representative tenants. Often these tenants can be divided into groups with similar requirements. The resulting features are specified in terms of domain-specific functionality and business rules at the abstraction level of the tenants.

However, two essential differences (relative to SPLE) must be highlighted that can be attributed to the multi-tenant SaaS context in which the feature models will be embedded:

1. The context of multi-tenancy and SaaS introduces *additional variability, especially driven by non-functional requirements*. Because there is a single deployment of the service line that is shared by all tenants and fully managed by the SaaS provider, tenants want to impose additional availability and performance requirements to gain more control over the delivered service. Furthermore, some tenants may require strict isolation of their data for security reasons, for instance in a separate database (or even a separate execution environment). This additional variability, which does not exist in traditional deployments, also needs to be explicitly represented in the feature model, more specifically in the form of service level agreements (SLAs).
2. Also, as an operational service line typically services many tenants simultaneously, actual down-time should be minimized. In the context of evolution, instead of undeploying (taking off-line) the service line or its features, a more preferred tactic is to deploy the updated features and reconfigure individual tenants to make use of these updated features. An essential enabler for this is *versioning support* in the feature model.

3.2. Service line architecture design and implementation

The next activities in the service line development process are the design of the service line architecture by the SaaS architect and the actual implementation of this architecture, including all variations, by the SaaS developers. During these activities, a number of important aspects have to be considered by the SaaS architect: (i) the use of architectural formalisms that support variability and variability management, and (ii) composability and traceability of individual features and their corresponding software variants in the service line architecture. These aspects *constrain* the architect in the following ways.

3.2.1. Architectural support for variability

The design of the service line architecture results in the typical architectural views (cf. Kruchten, 1995), showing the building blocks of the service line (structural view), the deployment of the service line (deployment view), processes and component behaviours (process view), etc. The service line architecture describes all possible variations in the multi-tenant SaaS application.

Specific variations are seldomly contained within one architectural view, especially when it comes to non-functional features (e.g. performance SLAs). For example, to realize performance-level variability, the architect has to introduce performance monitoring components in the structural view, model different resource allocations (e.g. nodes, CPU percentages, physical connections, etc.) in the deployment view, and introduce additional behaviour in the process view, e.g. to document the protocol to dynamically upscale. It is clear that when the amount of features increases drastically, documenting these variations in the complete architecture, in and across these architectural views will lead to complex, composed models in which it is hard to distinguish the base architecture from specific, individual variations.

Therefore, it is an essential decision in the creation of our method to express service line variations in separate *variability-supporting views*. These views enable the SaaS architect to separately manage and design different variations and versions in the service line separately. To realize this, the *variation points* in the service line as well as the *binding* between these and actual *variants* (i.e. alternative implementation artifacts) are therefore made explicit in the architectural description.

The current state of the art presents several (meta-models for) variability views that can be used in our method by the SaaS architect, for example Galster and Avgeriou (2012) and Groher and Weinreich (2012). However, these solutions lack support for a multi-tenant SaaS context, i.e. they do not offer support for (i) co-existing configurations that are activated on a per-tenant basis, and (ii) versioning to enable service line evolution.

3.2.2. Composability and traceability of features

At the level of the architecture, a feature is represented by a set of *(architectural) components*. Depending on the selection of middleware technology, such a component can be implemented by different implementation-level artifacts, such as classes, software components, web services and workflows, etc. It is an important concern during the development of service lines to ensure the *composability* of features and their corresponding software variants; i.e. to make sure that separately developed feature implementations remain compatible (syntactically and semantically), so that they can later be composed to a working system. To support this concern, the base architecture has to be built around stable interfaces (Van Landuyt et al., 2009), and components should be self-contained (modular) and preferably be stateless. The latter is especially relevant in a multi-tenant context because it simplifies dynamic customization (no need for quiescence) and thus limits the performance overhead. Modularity makes components interchangeable and enables SaaS developers to reuse components in different compositions and thus in the implementation of different features.

Furthermore, the SaaS provider should be able to associate the different variants in the service line architecture with the features in the feature model resulting from the domain analysis, but also with particular software artifacts in the implementation, and even at run time with the tenant-specific configurations and the dynamic compositions in the deployed service line. Maintaining these *traceability* links is thus crucial to combine the different activities of our method in an integrated process. Furthermore, rigorously documenting the traceability links between different artifacts will enable the creation of advanced management and development tools, for example to signal inconsistencies to the developer (e.g. when certain features exist in the feature model, but no implementation is available). In general, these concrete associations between features, architectural components and software artifacts enable the SaaS provider (i) to ensure consistency across the service line and to analyze the impact of changes (cf. Scenarios #1, #4, and #5), and (ii) to support self-service by automating the configuration of the service line based on the tenant-specific requirements (cf. Scenarios #2 and #3).

To achieve traceability, the SaaS architect has to explicitly define *feature-to-software-composition mapping* specifications for each feature in the feature model. In the state of the art, several approaches exist to support variability and traceability in software architectures (Groher and Weinreich, 2012) and throughout the process of software product line engineering (Bachmann et al., 2004; Cavalcanti et al., 2011). However, these approaches require some modifications to enable integration with our method, e.g. versioning support.

Notice that non-functional requirements do not necessarily map on particular software artifacts. For example, performance SLAs can

provide input for a monitoring service, and availability SLAs can be translated to a replication strategy. Although these features depend on the presence of particular middleware services, they do not necessarily have an impact on the service line implementation itself. In contrast, security is an example of a non-functional requirement that typically maps on particular components (e.g. for encryption, authentication or authorization).

3.3. Service line deployment and operation

After developing the service line, the SaaS operator creates an instance of the service line and deploys it on top of the cloud infrastructure of the SaaS provider. After this deployment process, the service line is accessible by the tenants (typically via the Internet) as a customizable, multi-tenant SaaS application.

The SaaS provider has to decide which environment is most appropriate to host the service line. For example, the SaaS provider can choose to deploy the service line on top of an existing PaaS or IaaS platform, or to set up his own cloud infrastructure. The former option requires no initial investment (i.e. infrastructure acquirement) and less maintenance, but the SaaS provider has less control over the cloud platform and thus needs to take these limitations into account. In addition, the necessary middleware support should be available or installed by the operator.

This process also includes *operating* the service line: aspects related to monitoring the running service line (e.g. to verify SLA compliance, to optimize resource usage across tenants, etc.), as well as managing and maintaining the different co-existing versions of the components, for example upgrading features, disabling certain variants, etc. (cf. Scenarios #4 and #5). Specific management tools can be envisioned to operate a running service line.

This selection of a suitable SaaS middleware platform that underpins the service line has a major impact on the architectural design, the implementation, as well as the deployment and operation process. We therefore have defined a feedback loop to the development process of our method (see Fig. 1). The middleware platform typically provides a number of enabling services for the development (e.g. component technology, libraries, etc.), deployment, management and operation of service lines. We require at least the following elements of the SaaS middleware:

1. *Support for versioning of features and feature implementations*. Versioning functionalities enable different versions of service line artifacts (e.g. modules, libraries, workflows, etc.) to co-exist. This is essential to realize, test and manage partial upgrades in service lines that potentially service a wide range of tenants. In addition, it supports the traceability concern. Furthermore, a common practice in cloud computing to reduce the risk of failure during upgrades, is to perform rolling upgrades (Dumitraş and Narasimhan, 2009). The SaaS middleware should offer the necessary support to enable such a gradual roll-out of upgrades.
2. *Basic support for multi-tenancy*. Multi-tenancy is a core characteristic of a service line. Guo et al. (2007) and Bezemer et al. (2010) discuss guidelines and approaches to develop a multi-tenancy enablement layer for multi-tenant SaaS applications. Such a middleware layer offers the necessary support for application-level multi-tenancy, for example to ensure data isolation between the different tenants. Each incoming request is associated with the corresponding tenant, and this context information is processed with the request throughout the SaaS application and the middleware platform (e.g. storage service). Some existing PaaS platforms (e.g. Google App Engine) already offer built-in support for tenant data isolation.
3. *Support for dynamic (run-time) composition*. Because a service line is inherently multi-tenant and thus shared by all tenants, the appropriate variants are activated at run time. Therefore, the implementation of the service line should support run-time (re-)composition. Our method is open for existing composition mechanisms, as the service line architecture makes abstraction of the heterogeneity of (composition) technologies provided by the underpinning platform. More specifically, SaaS developers should be able to combine multiple composition mechanisms in the service line implementation (e.g. both service and component composition). In the domain of adaptive and reconfigurable middleware, several technologies exist to support dynamic composition, for example using reflection and dynamic aspect-oriented programming (AOP) (Pawlak et al., 2001; Truyen et al., 2001; Popovici et al., 2003; SpringSource, 2013). However, these dynamic composition mechanisms should also be tenant-aware. For example, Walraven et al. (2011) support tenant-aware customization using dependency injection (Fowler, 2004), and Mietzner and Leymann (2008) focus on the customization of (BPEL-based) workflows based on tenant-specific requirements.

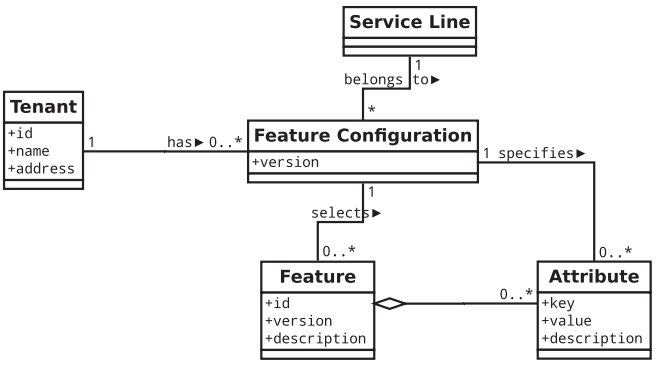

Fig. 2. Meta-model for tenant-specific feature configurations.

3.4. Tenant requirements analysis

During the activity of tenant requirements analysis, the tenant administrator is offered a configuration interface that enables the expression of the tenant's preferences and requirements. This configuration interface is based on the feature model that was created during the domain analysis activity (cf. Section 3.1). By selecting the appropriate features and configuring specific feature attributes, a *tenant-specific feature configuration* is created for each tenant. Because the feature model is specified in terms of domain-specific functionality and business rules, the tenant administrator does not need any technical knowledge and can thus be an employee of the tenant (thus enabling self-service). Existing tools for feature modeling often also support feature-based configuration (e.g. Kastner et al., 2009; Mendonca et al., 2009). Basically, such a configuration contains the IDs of the selected features.

The feature configuration meta-model describes the different concepts and their relations with respect to the tenant-specific configuration of service lines in terms of features (see Fig. 2). For each `Service Line`, a `Tenant` can specify a `Feature Configuration`. A `Feature Configuration` is derived from a feature model (not depicted in Fig. 2). It contains the set of features selected by the tenant administrator and, when applicable, specifies the value for a feature attribute to parameterize it. Similar to features, feature configurations are also versioned. This allows tenants to revert to earlier configurations when desired.

Finally, the feature selection process is subject to the constraints and dependencies as defined in the feature model. After the tenant administrator has specified the feature configuration, the relations between the different selected features are verified to ensure that no conflicts exist (e.g. using Binary Decision Diagrams like van der

Storm, 2007). Next, the feature configuration is used as input for the next activity of the service line configuration process.

When the tenant has new requirements (i.e. not yet supported by the service line (cf. Scenario #4 of Section 2.1)), a new development iteration of the service line might be started, potentially including updates to the feature model, the service line architecture and implementation, and finally updates to the service line deployment. Throughout the method, a number of software engineering principles have been applied to limit the potential impact of new requirements: (i) abstraction: because a feature model is hierarchical, lower-level details are typically handled near its leaves, and the coarse-grained structure of the feature model is expected to remain relatively stable, (ii) similarly, the base architecture is defined once and expected to remain stable over the life-time of the service line, while variations are described in separation of the architectural views describing the base architecture (cf. variability views), and (iii) fine-grained variations can lead to small version increments of individual components, and these can be activated at run time and on a per-tenant basis.

Evidently, the impact of new requirements will depend highly on the nature of the new requirements. In Section 5.2, we further discuss how new requirements can be dealt with in the context of service lines.

3.5. Configuration mapping

In traditional SPLE, a feature configuration leads to the instantiation of a specific product instance from the product family. Although the feature configuration is a key artifact in the process to instantiation of a single product, it is not visible in the end result (i.e. it has been compiled away during the process). In the context of multi-tenant SaaS applications however, a tenant-specific feature configuration (cf. Fig. 2) has to be translated to an actual *software configuration* that specifies the composition of architectural components in the service line (as expressed in the variability views). These tenant-specific software configurations co-exist in the running service line. Not only will different tenants have different software configurations, but also different configurations of a single tenant may co-exist to reflect e.g. different versions, or different choices for different end users. Maintaining these co-existing configurations introduces substantial management overhead.

The next activity in the configuration process therefore entails the *automated transformation* of a tenant-specific feature configuration to a software configuration. A software configuration defines for a particular tenant which specific variants should be bound to the different variation points. Features are thus translated into a corresponding configuration of software artifacts. Automation of this activity is possible by applying the mapping specifications as defined during the design of the service line architecture (see Section 3.2). This automation is essential to preserve cloud scalability and to ensure consistency.

Before activating the automatically generated software configuration, an optimization step might be required. Architectural components can have multiple variation points, which can be fulfilled by different features (i.e. by specifying bindings). Therefore, multiple occurrences of the same architectural component in a software configuration have to be eliminated by integrating and combining the different bindings. This optimization step can be incorporated as part of the automated transformation.

3.6. Configuration activation

While in the application engineering phase of the SPLE method a dedicated product instance is delivered and deployed for each tenant, the service line composition activity involves adapting the service line at run time to match the tenant-specific software configuration. Concretely, after the tenant administrator has specified a feature configuration, the automatically generated software configurations are immediately effective and the end users associated with that tenant can start using the service line application.

At run time, it is decided on a per-request basis which software variants should be activated, depending on the configuration of the tenant associated with the current request. As mentioned in Section 3.3, the service line (or the middleware on top of which it is built) should offer both explicit support for multi-tenancy and dynamic composition to realize this.

In a typical implementation strategy for tenant-aware dynamic composition (e.g. Walraven et al., 2011), the service line middleware intercepts each incoming request and links it to the appropriate tenant (via the associated context information). Next, the corresponding software configuration is fetched. Each time a variation point is reached during the execution of a request, the appropriate variant, and thus software artifacts, are dynamically composed into the service line.

4. Service line engineering in practice

The service line engineering (SLE) method presented in this paper has been realized in the context of a collaborative research project with industry, called CUSTOMSS (CUSTOMSS, 2011). The consortium of this project comprises of three European B2B SaaS providers, active in the domains of health-care, medical image processing and document processing. Although we applied the SLE method and its principles in all three SaaS applications, the main validation was conducted in the context of document processing in close collaboration with the SaaS provider. We introduce this application in Section 4.1.

We adopted a prototype-driven validation approach, in the sense that we applied our SLE method to develop a prototype (that represents a significant subset of the entire document processing SaaS application), and we leveraged the resulting prototype to validate the practical feasibility and applicability (in this section), and to evaluate the benefits of SLE (in Section 5).

Sections 4.2–4.5 present an activity-per-activity account of how we applied the SLE method to develop the prototype. For each activity, we provide the relevant artifacts, and briefly discuss the key design decisions. During development, we had to bridge some gaps by complementing our generic method with some auxiliary elements to make it effective in practice: (i) a variability metamodel that supports co-existing configurations, (ii) an approach to define reusable feature-to-software-composition mappings, and (iii) a basic set of middleware services to facilitate the development and management of service lines. Throughout this section, we highlight these auxiliary elements.

4.1. Document processing SaaS application

UnifiedPost[1] is a European SaaS provider that offers B2B document processing facilities to a wide range of companies in very different application domains. This multi-tenant SaaS application supports the creation and generation, the business-specific processing and the storage of millions of business documents per day, such as invoices and payslips, even up to printing and distributing. This is a fairly large-scale and complex SaaS application, that currently services around 150 different tenant organizations. In the current operation, UnifiedPost distinguishes between roughly 25 clusters of tenants, grouping companies that have

[1] http://www.unifiedpost.com/.

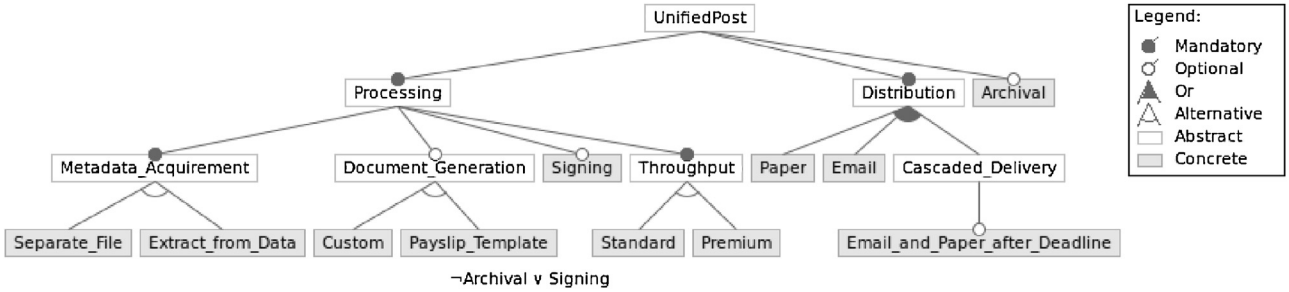

Fig. 3. Feature diagram showing a simplified subset of the variations in the document processing case study. Specifically, only the relevant features for the two tenants presented in Section 4.1 are depicted. The diagram is created using FeatureIDE (Kastner et al., 2009).

reasonably similar requirements. These clusters all represent different high-level and parameterizable variations of the application.

The prototype focuses on servicing one of these tenant clusters and in the next paragraph, we present two examples of tenant companies[2] in that cluster that have different document processing requirements. For readability, we elaborate on a limited part of the UnifiedPost application, and we show only a fraction of the existing variations. Nonetheless, these are sufficiently representative examples of realistic variations within one cluster of tenants of the document processing SaaS application.

Due to space limitations, this section only presents a strongly summarized account of the entire validation effort. Please refer to Walraven et al. (2013) for a detailed account of applying our method to develop the document processing SaaS application, as well as for further details on the auxiliary solutions that we developed. The source code of the prototype, including all feature mapping specifications, and the supporting middleware services is publicly available.

Running example. Tenant A is a temporary employment agency that uses UnifiedPost's document processing application to process the payslips of all its employees. On a regular basis, Tenant A provides a set of payslip documents, along with some meta-data, and requires the SaaS application to print the payslips and distribute them among the different employees based on the associated meta-data. Since Tenant A has a large amount of employees, it has strict demands in terms of the guaranteed throughput of the document processing application, and it is prepared to pay premium fees to obtain a specific SLA in terms of throughput and deadlines.

Tenant B is active on the financial services market and uses the SaaS application for processing its invoices and distributing these to corporate customers. Tenant B only provides raw data (i.e. document data and meta-data) as input, for example in a spreadsheet or in CSV format, and thus requires the generation of the invoices (based on a custom layout). The generated documents should be delivered to the customers (i.e. end users) of Tenant B, depending on their preference: via email or on paper (printed mail). In the case of email delivery, a link to the document is provided, which enables the customer, after authenticating, to view and download the document. However, if after 48 h the document has not been retrieved, the particular document should still be printed and sent via printed mail. In addition, all generated documents should be signed and archived securely for a legally defined period (e.g. 24 months). Tenant B does not want to pay an extra charge for a guaranteed throughput level.

4.2. Domain analysis

Fig. 3 presents a subset of the feature model that is the main result of the domain analysis activity. This is a simplified feature diagram that covers the subset of the UnifiedPost application that matches the running example.

At the top level, the service line offers features related to `Processing`, `Distribution` and `Archival`. `Processing` groups the following sub-features: (i) different alternatives to acquire metadata from the raw input data (under `Metadata_Acquirement`), (ii) an optional feature for document generation based on different templates (under `Document_Generation`), (iii) an optional feature named `Signing` for signing documents using the SaaS provider's or tenant's certificate, and (iv) performance SLAs (`Throughput`) that offer different performance levels.

In terms of `Distribution`, the tenant can select from several (single) delivery channels (i.e. email or printed mail), or he can opt for a cascaded delivery. In the latter case, first an email is sent out, and if the document is not retrieved after a certain period, then the document is printed and sent on paper.

Finally, the optional `Archival` feature enables tenants to archive documents conform legislation. The logical expression at the bottom of the figure ($\neg Archival \vee Signing$) represents a dependency between features: documents can only be archived if they are signed. So, selecting the `Archival` feature implies that the `Signing` sub-feature must be selected as well.

We used FeatureIDE (Kastner et al., 2009) as modeling tool to create this feature diagram, which does not support all required model elements for our method (e.g. attributes and versioning). However, these elements do exist in the underlying model specification in our prototype.

4.3. Service line architecture design and implementation

Below, we discuss how we designed and implemented the document processing service line. More specifically, we focus on illustrating how we realized variability in the architecture (modeling specific variations) on the one hand, and the composability and traceability of features in the design and implementation of the service line on the other hand.

Variability meta-model. We designed the document processing application as a customizable workflow (i.e. a service orchestration) expressed in the Business Process Modeling Notation (BPMN). The workflow is triggered when raw data is uploaded by the tenant. In the first step of this workflow, metadata is acquired from this raw input data. Subsequently, the output documents are (optionally) generated based on this metadata and according to the template selected by the tenant. Then, the generated documents are delivered to the appropriate recipients, using the selected delivery mechanism.

Variability is required in the application at two different levels of abstraction. At the workflow level, the abstract service types (e.g. delivery channel for documents) represent the variation points, and each service implementation (e.g. email delivery service) represents a specific variant. Thus, the composition type is a regular

[2] For non-disclosure reasons, we anonymized the names of the tenant companies.

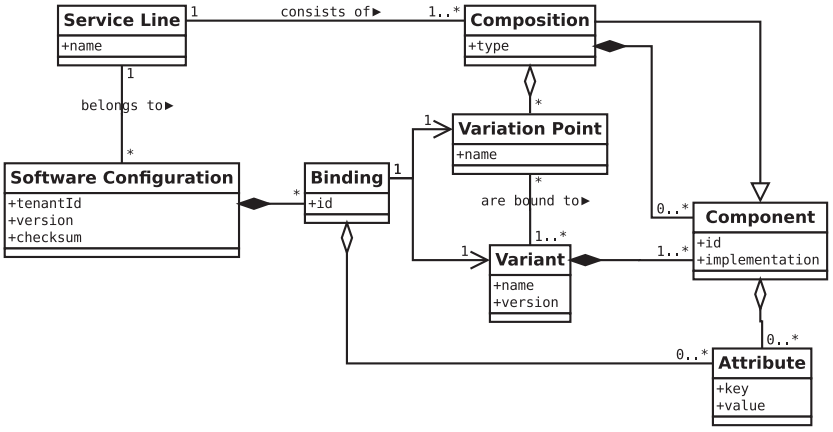

Fig. 4. Meta-model for variability in service line architectures.

service composition. Variability is also required at a more fine-grained level, i.e. at the level of individual service implementations. For example, the document generation service offers several strategies for document generation, and in this case, we use dependency injection as composition type.

In its architectural design, the document processing workflow thus comprises a basic flow (the mandatory features) and a set of extensions, which are developed as separate, modular entities. To realize this separation, the workflow complies to the generic variability meta-model presented in Fig. 4. This meta-model stipulates that (i) the service line is described as a set of compositions (`Composition`), and (ii) there is no fixed level of granularity at which variability can be supported (represented by the composite structure around `Component` and `Composition` which allows the service line to be structured hierarchically). By specifying explicit `Variation Point`s for a `Composition`, the architect can indicate whether that particular composition is customizable.

A `Service Line` is a set of compositions, some of which are customizable. The meta-model therefore allows variations to be introduced at different levels of abstraction: from the top level of coarse-grained system-level components, down to the level of fine-grained compositions. Furthermore, the meta-model makes abstraction of the specific composition mechanisms and technologies (cf. the `type` attribute) that will be used to compose these variations at run time.

This degree of openness is an essential requirement to support customization of multi-tenant SaaS applications. Variability in multi-tenant SaaS applications is typically not limited to one specific level of granularity, and SaaS middleware commonly provides a range of complementary run-time composition and adaptation mechanisms.

Furthermore, the support for co-existing configurations across the different tenants is an important requirement to enable the customization of multi-tenant SaaS applications. Therefore, the meta-model supports multiple co-existing `Software Configurations`, which define for a particular tenant (cf. the `tenantId` attribute) which specific variants should be dynamically bound to the different variation points. Such a software configuration represents the result of the automatic transformation of tenant-specific feature configurations (cf. Section 3.5). Since a tenant can have multiple versions of his feature configuration, software configurations also have to be versioned to enable the tenant to easily compare with an earlier version.

The openness of the proposed variability meta-model with respect to (i) the level of granularity of software variants and (ii) the technology that is used to realize the corresponding compositions, as well as (iii) the support for co-existing configurations, reaches beyond the state of the art in architecture-level variability views (cf. Section 6.3).

Listing 1
Example of a feature mapping for the Cascaded Delivery feature.

```
1   featuremapping CascadedDelivery-Email-Paper-Deadline {
2     feature Email_and_Paper_after_Deadline-v1;
3     composition DocumentProcessing {
4       binding {
5         variationPoint: DeliveryChannel;
6         variant: CascadedDeliveryWorkflow-v1;
7       }
8       composition: CascadedDelivery;
9     }
10    composition CascadedDelivery {
11      binding {
12        variationPoint: FirstChannel;
13        variant: EmailService-v1;
14      }
15      binding {
16        variationPoint: SecondChannel;
17        variant: PostalService-v1;
18      }
19      binding {
20        variationPoint: DecisionAlgorithm;
21        variant: TimeoutBasedDecisionAlgorithm-v1;
22        attribute: timeout;
23      }
24    }
25  }
```

Mapping features to variants. In order to enable the automated and dynamic composition of specific variants, machine-interpretable mappings between the features and the software variants have to be defined. In addition to enforcing a hierarchical design structure, the meta-model of Fig. 4 allows the SaaS architect to define feature-to-software-composition mappings. These mappings bind a `Variation Point` to a specific `Variant`. To realize a `Binding`, in some cases specific attributes (`Attribute`) have to be filled in.

We have specified a mapping for each feature in the document processing application (i.e. those defined in Fig. 3) to components and compositions defined in the service line architecture, using a custom grammar as presented in Walraven et al. (2013). For example, Listing 1 shows a feature-to-software-configuration mapping that specifies which variants to use in case the `Email_and_Paper_after_Deadline` feature (line 2) has been selected by the tenant. The workflow for the document processing application is defined as the `DocumentProcessing` composition. A relevant variation point for the running example is the delivery channel (line 5), and the

`CascadedDeliveryWorkflow` (using multiple delivery channels) is the corresponding variant for this feature (line 6). This variant, however, consists of another (customizable) composition, called `CascadedDelivery`. Therefore a dependency is added to this composition via its ID (line 8). The latter composition has two variation points for the two possible delivery channels (lines 11–18), and a variation point for the algorithm that decides when to use the second delivery channel. For this feature, the timeout-based decision algorithm is selected (line 19–23). In addition, the timeout attribute in the `Email_and_Paper_after_Deadline` feature is passed through as input for the decision algorithm via the 'timeout' attribute in the mapping (line 22).

The proposed approach to define reusable feature-to-software-composition mappings that can be applied to automatically (re)configure the multi-tenant SaaS application at run time is an important new element in our solution (cf. Section 6.3).

4.4. Service line deployment and operation

Our prototype implementation is a layered solution. In a first step, we developed a reusable middleware layer to support the management of service lines, on top of which we implemented the application-specific document processing services.

Service line management support layer. In addition to the enabling middleware services that address the core requirements of (i) versioning support, (ii) application-level multi-tenancy and (iii) dynamic composition (cf. Section 3.3), we have developed a generic support layer to facilitate the development and management of service lines (Walraven et al., 2013).

This layer offers several service interfaces to the different stakeholders and to the application layer. The `IFeatureManagement` interface provides the SaaS architect/developer with a service to manage the feature model and the feature mapping specifications. Tenant administrators, which are (in the context of service lines) employees of the tenant organization (cf. self-service in Section 3), have access to the `ITenantManagement` interface. This interface enables the registration of new tenants to the service line as well as the tenant-specific customization by selecting features based on the tenant's preferences and by parameterizing these features. The internal `ConfigurationMapping` service ensures that feature configurations are automatically transformed into tenant-specific software configurations by means of the feature mapping specifications. Finally, the `ITenantConfigurationRetrieval` interface is offered to the multi-tenant application, more specifically to allow the look-up of tenant-specific software configurations and to enable the run-time composition of the SaaS application (cf. Section 3.6).

We have implemented these services as a reusable middleware layer using Java EE. The front-ends are realized with Java Servlets, JSPs, and RESTful services; the business logic is implemented using session beans; for persistence we used entities that are stored into a MySQL database. Furthermore, the custom grammar to express the feature-to-software-composition mappings is developed using ANTLR. This middleware layer is designed for reuse across different service lines.

Prototype implementation. The document processing prototype is built on top of JBoss Application Server 7, which is Java EE 6-compliant. It uses jBPM and Drools for customizable document processing workflows, and MySQL for persistence. To achieve application-level multi-tenancy and tenant-aware dynamic composition within services, we rely on the modular middleware layer that we presented in previous work (Walraven et al., 2011). To realize the throughput SLAs in the running example, we have provided alternative service implementations that each guarantee a certain throughput (premium versus normal).

4.5. Tenant provisioning

As indicated in Section 3, the actual provisioning of (new) tenants becomes relatively straightforward by leveraging on the investment of service line development and deployment. During the Tenant Requirements Analysis activity, tenant administrators use the `ITenantManagement` configuration interfaces presented above, to select and parameterize features based on their requirements. For *Tenant A* and *Tenant B* of the running example, this configuration activity has resulted in the following two feature configurations:

```
Tenant A:                 Tenant B:
  Separate_File             Extract_from_Data
  Premium                   Custom
  Paper                     Signing
                            Standard
                            Email_and_Paper_after_Deadline timeout:48
                            Archival duration:24
```

These feature configurations specify the set of selected features, and optionally the associated attributes and their values. The validity of these configurations are checked by looking at the constraints (mandatory features, dependencies, etc.) defined in the feature model (cf. Section 4.2). This way, errors and invalid configurations can be avoided before activation.

Subsequently, these feature configurations are automatically transformed into tenant-specific software configurations (cf. Configuration Mapping activity). Because of the automation, the effort associated with this activity is minimal and the updated configurations are thus immediately effective for the end users of the tenants.

Finally, the composition of particular services into the document processing workflow occurs at run time based on the tenant-specific software configuration that is applicable to the incoming request (cf. Configuration Activation activity). For example, when a document processed for *Tenant A* is ready to be delivered to the appropriate recipient, the service line will activate the `PostalService` variant for the `DeliveryChannel` variation point (corresponding to the `Paper` feature). The document is first sent to the printing service and next the printed document is delivered via the postal service to the appropriate recipient's address. In comparison, a delivery workflow that is started for *Tenant B* (cf. Listing 1) sends the output documents via email to the recipients. When a particular document has not been fetched after a timeout period of 48 h, it is also printed and delivered via the postal service.

These two different tenant-specific configurations of the document processing service line regarding document delivery are activated at run time and can execute simultaneously. Further details about the run-time customization of the document processing service line can be found in Gey et al. (2013).

5. Evaluation

This section evaluates our service line engineering (SLE) method as follows. We perform a comparative cost analysis of the required effort in each of the scenarios of Section 2.1 and substantiate the claimed efficiency benefits in Section 5.1. This section ends with a discussion of the strengths and limitations of the presented method in Section 5.2.

5.1. Service line efficiency

In Section 2.1, we presented five key development and management scenarios that occur in the life-cycle of a multi-tenant SaaS application. In this subsection, we re-evaluate these scenarios in

the context of our SLE method. Specifically, we compare the current state of practice (discussed in Section 2.1) with our SLE method *by analyzing the cost for the SaaS provider (in terms of required effort)* to perform these scenarios. We leverage on this cost analysis to substantiate the claimed efficiency benefits. The main cost variables that affect the required effort are t (the number of tenants), f (the number of features), and v (the average number of variation points per feature). The different costs are functions of these variables and they grow with an increasing value of these variables.

First, we look at the cost to provision (new) tenants with a configuration (cf. Scenarios #1 to #3). Next, we discuss the cost related to evolving and updating the service line itself (cf. Scenarios #4 and #5).

5.1.1. Provisioning new tenants

The cost to provision (new) tenants with a configuration has a major impact on SaaS scalability: if the provisioning cost grows significantly with an increasing number of tenants, then this hinders scalability of the SaaS offering. We discuss for each relevant scenario the associated cost for the SaaS provider.

Scenario #1: Initial development of SaaS application. Our SLE method clearly requires more effort in the initial development phases: variability analysis has to be performed, leading to the creation of an initial feature model. In addition, the feature-to-software-composition mappings between features and software artifacts have to be specified explicitly. This extra cost compared to the current state of practice thus mainly depends on the number of features f and variation points per feature v.

Scenario #2: Provision a new tenant. In the current state of practice (see Section 2.1), the SaaS provider has to (manually) specify and test a tenant-specific software configuration for f features and an average of v variation points per feature that have to be bound to a particular variant.

With the SLE method however, once the service line is up and running, we can rely on self-service and automation and so provisioning new tenants does not require any intervention from the SaaS provider. Specifically, the tenant administrator (i.e. employee of the tenant) can now customize the SaaS application by selecting and configuring the desired features via a management interface. Subsequently, a tenant-specific software configuration is generated in a consistent way and – given that the feature mapping specifications themselves are well-tested and correct – free of errors. Thus, with respect to provisioning new tenants, the SLE method results in a constant cost for each tenant and no extra cost in terms of required effort for the SaaS provider.

Scenario #3: Update tenant-specific configuration. A service line allows the tenant administrator to update the tenant's feature configuration via the management interface. Based on this updated feature configuration, a new tenant-specific software configuration is generated, which is immediately effective for the service line, without any (human) intervention. Essentially, the cost of updating a tenant-specific configuration is equal to the cost of provisioning a new tenant (see Scenario #2).

Conclusion. This cost analysis indicates the trade-off between spending effort on early development of the service line versus the costs and effort to configure and compose the running service line after deployment. The efficiency benefits are realized by leveraging service line design elements (feature mappings, composability of features, feature traceability, modularity of feature implementations, etc.) that have been created (and thus incurred an extra cost) during the initial development of the service line.

In the current state of practice, the effort to manage the running SaaS application will grow drastically with an increasing number of tenants t, which hinders the scalability of the SaaS offering. In comparison, because of the automation of most activities in the configuration and composition processes, the increase of effort in the service line engineering method will largely depend on the number of features f and not the number of tenants. Since the number of tenants is typically a magnitude larger than the number of features and the cost to specify a feature mapping is generally smaller than the cost to create an entire software configuration,[3] the higher initial effort required by our method to develop the service line is rapidly reversed and more importantly will enable the SaaS provider to realize the benefits of scale required to run a profitable SaaS offering.

Especially the required effort to realize Scenario #2 (i.e. provisioning new tenants) is an important metric, as it expresses the time to service and provision new tenants. A faster time-to-provision will provide the SaaS provider with a real business advantage over its competitors.

5.1.2. Maintenance and evolution

Evolution entails introducing new requirements as well as updating and maintaining multi-tenant SaaS applications. Moreover, in SaaS applications updates can be performed more frequently, requiring the SaaS provider to adopt a smoother, more gradual strategy of continuous evolution. Because of this continuous flow of updates, the cost of evolution also has an important impact on SaaS scalability and profitability. We discuss the initial analysis of the cost of evolution based on Scenarios #4 and #5.

Scenario #4: Support new requirements. In the current state of practice, new requirements incur a heavy cost for verifying and (manually) updating *each* tenant-specific software configuration with f features and an average of v variation points per feature (see Section 2.1).

In a service line, new and unforeseen requirements will affect the feature model (cf. the feedback loop in Fig. 1). For example, a new top-level feature could be introduced or an existing feature could be further decomposed into one or more sub-features. Adding a feature requires extending the feature model and the service line architecture, as well as specifying a feature mapping for the new feature. This will not (or hardly) affect existing features, mapping specifications and feature implementations thanks to the composability of the software variants and the reusability of the feature mappings. However, when adding new features, the existing tenant-specific configurations must be verified and potentially updated to fix compatibility issues.

In summary, the cost to add a new feature includes the effort to create a new feature mapping that defines v variation points, and a constant cost per tenant to verify his feature configuration and to handle issues. We consider it to be a constant because a service line offers automated support for verifying whether existing feature configurations comply to the constraints of the updated feature model and for localizing issues. This enables the SaaS provider to quickly detect any issues and handle them (e.g. by setting a default feature for a new variation point, or by supporting multiple co-existing versions of the service line), independently of the size or complexity of the feature configuration.

In general however, the effort to support new requirements can be significantly higher, especially when these involve major changes in the feature model and the service line architecture. However, the traceability links between features, architectural components and software artifacts enable the SaaS provider to quickly identify and manage these interactions, and the modularity of the architectural components will limit the extent of ripple

[3] This is confirmed in our prototype: the average size of a mapping specification is around 12 lines, while the average size of the generated software configurations is around 44 lines.

effects throughout the service line architecture and implementation (cf. Section 3.4).

Scenario #5: Update and maintain SaaS application. Using our SLE method, this scenario only involves updating the feature mapping specifications instead of the entire configuration of *all* tenants (cf. Section 2.1). Because of the traceability support in the service line architecture, the SaaS provider can easily identify which specific feature mappings have to be updated, and a smaller amount of mappings have to be updated instead of all of them (i.e. *f*).

In case the updates conflict with existing tenant-specific requirements, this problem will be detected during the update process of the mapping specifications. The tenants to which such conflicts apply can easily be identified, i.e. these are the tenants who selected this particular feature in their configuration. In this case, the SaaS operator can decide to perform a gradual roll-out of the updates, temporarily hosting the old as well as the new versions of the features and software variants. Based on the version information in the tenant-specific configurations and the feature mapping specifications, the appropriate version will be activated in the service line at run time.

Conclusion. We have shown that the use of a feature model and feature mappings is also beneficial to evolving and updating the service line itself. Many changes can be encapsulated within the feature model and the feature mappings, while the current state of practice requires the SaaS provider to verify and update *all* configurations (because of the ripple effects). Similar to the proportion between the effort required for creating a feature mapping and for creating a tenant-specific software configuration, the effort for updating a feature mapping is in general also smaller than the effort for updating an entire software configuration. As a consequence, the higher initial effort during development also enables the SaaS provider to realize benefits of scale while evolving and maintaining the running service line.

Moreover, the SaaS paradigm enables SaaS providers to update their application continuously and rapidly. Consequently, the initial development effort is a one-time (or infrequent) investment cost compared to the continuous updates ("develop once, adapt/evolve forever"), which results in major cost benefits in the long term w.r.t. the management and customization of multi-tenant SaaS applications.

5.2. Discussion

This section provides an open-ended discussion on the strengths and limitations of the service line engineering method presented in this paper. In parallel, we discuss future work.

Non-functional tenant requirements. As indicated in Section 3.1, the context of multi-tenant SaaS introduces additional variability, especially driven by non-functional requirements (or quality attributes) such as availability, performance and security. These requirements are represented as SLAs in the feature model. In the other activities of our method however, these non-functional requirements also have to be taken into account: quality attributes are often important architectural drivers and put constraints on the deployment environment. For example, a monitoring service is necessary to monitor the resource usage on a per-tenant basis, while a policy enforcement engine has to ensure that the delivered performance is in compliance with the tenant-specific SLAs. These enabling middleware services are identified during the design of the architecture and should be provided by the underpinning platform.

Furthermore, a SaaS application can depend on third-party services. For example, in the document processing application, the SaaS provider could decide to outsource the distribution of printed documents. Even in such a cross-organizational context, the different SLAs still have to be ensured. In the state of the art, several solutions exist for service-oriented product lines to negotiate with third-party services to provide quality-of-service (QoS) guarantees, e.g. Kotonya et al. (2009) and Lee and Kotonya (2010). It is worth to investigate the applicability of these approaches in a multi-tenant SaaS context.

Our mapping specification (see Section 4.3) focuses on features (and SLAs) that can be mapped to a set of architectural components (and their corresponding software artifacts). With availability and performance SLAs this is often not the case. These features (and their attributes) provide input for the underpinning middleware services or a broker. In future work, we will extend our middleware support to include these SLAs in the automated configuration mapping activity.

Support for evolution and maintenance. Although our method covers the evolution and maintenance aspects of service lines, we did not provide specific service management functionalities to support the SaaS provider with the evolution and maintenance of service lines. For example, additional tool support is required to enable the SaaS provider to analyze the impact of an update on the service line architecture and the different tenant-specific configurations. In addition, since multiple versions of the features, configurations and software artifacts can co-exist, the SaaS provider requires explicit support to monitor and manage (i.e. upgrade, disable, make obsolete, etc.) these versions within the running service line. This middleware support is currently lacking in our implementation. In fact, most of these issues are open challenges for SaaS applications. In future work, we will further investigate these issues to support effective operation of service lines.

Besides the impact analysis of an update based on the service line architecture and the different configurations, updates have to be tested and validated thoroughly before they are rolled out. This is especially relevant in a multi-tenant SaaS context, because conflicts or errors (e.g. caused by updates) potentially affect all tenants and thus the availability of the SaaS application. To realize continuous and seamless updates to service lines (part of service line operation), there is a need for integration with frameworks for automated testing and gradual roll-out of updates. These requirements are out of scope for this paper, but are relevant for a service line engineering method to be investigated in future work.

Finally, it is possible that during the tenant requirements analysis new requirements are discovered that cannot be addressed. For example, the new variations that have to be introduced cannot be implemented without run-time conflicts. Note that this issue is not limited to service lines, as it can also occur in traditional software product lines. However, multi-tenant SaaS applications have the additional constraint that these different requirements should be addressed by a single application instance and customization should be achieved by means of dynamic composition. When the requirements cannot be met because of this multi-tenant context, the SaaS provider could decide to deploy and operate a second (different) instance of the service line. However, this requires the SaaS architect to make a distinction between deploy-time and run-time variability (e.g. by using variability types like Galster, 2010), and multiple deployments evidently result in higher resource usage. Therefore, the SaaS provider has to make a comparative assessment (ability to address more requirements versus increasing management complexity and resource usage).

6. Related work

This section discusses three categories of related work: (a) dynamic and service-oriented product lines, (b) customization of multi-tenant SaaS applications, and (c) variability management in software architecture.

6.1. Dynamic and service-oriented product lines

The work on service-oriented product line engineering (SOPL) (Cohen and Krut, 2008; Günther and Berger, 2008; Medeiros et al., 2009) tries to combine the benefits of the open-ended model with late-bound variability in service engineering (SE) with the closed world of managed variability and planned reuse in SPLE. The fundamental building block of a service-oriented architecture (SOA) is a service; the application (i.e. the product instance of a SOPL) is thus an orchestration of services. However, in the case of SaaS, the application itself is a service. The key difference however remains that a SOPL creates different product instances per customer that have to be deployed separately by the customer (possibly in an automated way). Although a (multi-tenant) SaaS application can be developed based on a SOA, a service line results in a single, multi-tenant product instance (deployed and managed by the SaaS provider) that is dynamically customized based on the tenant-specific configurations.

Nguyen et al. (2012) try to increase the cost efficiency of customizing web services: their work keeps track of which service variants are already deployed to ensure only one instance is deployed per variant. When compared to application-level multi-tenancy, this approach is less efficient in terms of operational costs (resource usage and maintenance effort), especially when many variants exist (i.e. high amount of features). Therefore it is not suitable for the development and customization of multi-tenant SaaS applications.

Furthermore, the concept of SOPL is often combined with dynamic software product lines (DSPL) (Hallsteinsen et al., 2008; Baresi et al., 2012) because of the open-ended model of services: services can be discovered and consumed at run time (Hallsteinsen et al., 2009; Kotonya et al., 2009; Lee and Kotonya, 2010). In addition, a software system can use a DSPL to cope with changes in the current context or environment, or with unforeseen situations (e.g. Parra et al., 2009; Baresi et al., 2012). Although these approaches cope better with run-time variability, these dynamic adaptations are applicable to all users of the product instance (instead of on a per-tenant basis). There is no support for co-existing (tenant-specific) configurations. Moreover, a DSPL is still limited to the features that were included during feature selection.

Clearly, the current SOPL approaches are based on traditional, static SPLE approaches, which are not suited for multi-tenant applications. Our service line engineering method, however, focuses on the development and customization of multi-tenant SaaS applications to efficiently manage and reuse the different software variations and configurations, even at run time.

6.2. Customization of multi-tenant SaaS

Mietzner et al. (Mietzner and Leymann, 2008; Mietzner et al., 2009) have extensively studied the customization of multi-tenant SaaS applications. They apply variability modeling techniques from the SPLE domain to support the management of variability in service-oriented SaaS applications, more specifically BPEL processes in the cloud. The variability is realized by defining variability descriptors that create application templates. Tenants fill in these application templates to create tenant-specific BPEL processes, which have to be deployed separately. Our work, however, introduces an integrated method for the development and customization of multi-tenant SaaS applications, independent from the underpinning technologies. In addition, we provide support for the management of the different software variations and tenant-specific configurations. The solutions presented by Mietzner et al. can be integrated in our work as a possible tenant-aware customization technique for workflows (instead of the current solution in our prototype using jBPM and Drools).

In previous work (Walraven et al., 2011), we have developed a middleware layer for PaaS platforms to enable tenant-specific customizations, while limiting the performance overhead and preserving the operational cost benefits. It focuses on SaaS applications consisting of a single service and dynamic composition is supported via dependency injection. As it provides a tenant-aware customization technique, this middleware layer could be one way to realize the service line composition activity of our method (as illustrated by our prototype).

Schroeter et al. (2012a) identify requirements for a variable architecture for multi-tenant SaaS applications and describe how their existing architecture for self-adaptive systems can be extended to support multi-tenancy. In contrast, the focus of our work is on providing an integrated method for efficient customization of multi-tenant SaaS applications and is to a large degree independent from any specific system or platform.

In Schroeter et al. (2012b), the authors propose a concept for dynamic configuration management, with different stages (Czarnecki et al., 2005) and stakeholder views in the configuration phase. Their work is limited to the problem space and covers only one activity of our method, i.e. the tenant requirements analysis. In this paper, we focus on the tenant and the SaaS provider, but our approach does not exclude more stakeholders and stages and is thus complementary.

Furthermore, work has been performed in the context of architectural patterns for developing and deploying customizable multi-tenant applications. Mietzner et al. (2011) present several multi-tenancy patterns and describe how services in a service-oriented SaaS application can be composed and deployed using these different multi-tenancy patterns. In this paper we focus on customization in application-level multi-tenancy, which maps to their single configurable instance pattern. Kabbedijk et al. (Kabbedijk and Jansen, 2011; Kabbedijk et al., 2013) present architectural patterns to realize run-time variability in multi-tenant SaaS applications. These patterns are relevant during the service line development process (Activity 2) to implement composable variants. For example, our dynamic composition technology (Walraven et al., 2011) corresponds to the Component Interceptor Pattern.

6.3. Variability management in software architecture

Several solutions exist for the representation and management of variability in service compositions, mainly limited to providing extensions to the BPEL language (Nguyen et al., 2011). Abu-Matar and Gomaa (2011) introduce a multiple-view modeling approach for variability in service-oriented architectures (SOA) in a platform-independent way. However, their approach focuses on a SOA context with views for business processes and service interfaces, and it does not support dynamic composition. Therefore, it is not directly applicable in a multi-tenant SaaS context.

Galster (2010) suggests a more generic approach for representing variability in SOA. However, their meta-model inherently assumes multiple instantiations of a SOPL, and thus not supports co-existing configurations. In addition, it does not associate variation points and variants with software artifacts. A more extended meta-model of an architectural viewpoint for variability is proposed by Galster and Avgeriou (2012). Such a variability viewpoint facilitates the representation and analysis of variability in software architectures. The proposed variability viewpoint is complementary to our method and could be integrated in the service line architecture, but this requires adapting our feature mapping specification. In addition, the meta-model of Galster and Avgeriou (2012) would have to be extended with versioning support and with the concept of (tenant-specific) configurations.

Groher and Weinreich (2012) integrate variability management support into an existing approach for describing and analyzing

software architectures. This results in one consistent model which links architectural artifacts (e.g. requirements, features, components) with the variability models to support full traceability. Our work is complementary, as we defined an integrated method for service line engineering that depends on variability management and traceability to support efficient reuse of software variations and configurations. Therefore, provided some modifications, the solution by Groher and Weinreich (2012) could be a suitable enabler to implement our method. Again, their solution would have to be modified to support versioning, and thus to support co-existing configurations and evolution of service lines.

Furthermore, Bachmann et al. (2004) and Cavalcanti et al. (2011) present meta-models to support variability and traceability throughout the process of software product line engineering (SPLE). As above, this work is complementary to our method, more specifically the service line development process, provided some modifications regarding the multi-tenant SaaS context.

Finally, Hubaux et al. (2011) support multiple perspectives in feature-based configuration. Configuration views are tailored to the profiles of the various stakeholders. In this paper, we have considered a limited number of stakeholders (employees of the tenant and the SaaS provider). However, other stakeholders could be involved in the tenant requirements analysis, for example a service reseller. Therefore, this approach of multi-view feature-based configuration is complementary and can be plugged into our service line method, more specifically as part of the tenant requirements analysis activity (see Section 3.4).

7. Conclusion

This paper presents an integrated service line engineering method for multi-tenant SaaS applications. This feature-oriented method supports co-existing tenant-specific configurations, and facilitates the development, deployment, run-time configuration and composition, operation and maintenance of SaaS applications, without compromising the scalability. The method is generic in the sense that each activity is open for existing work to be leveraged upon, yet it imposes some specific constraints (e.g. composability and traceability of features) and some enablers (e.g. multi-tenancy, versioning support, dynamic composition) to obtain the desired variability in the service line.

We have validated this method in the development of a service line for a real-world industrial SaaS application in the domain of document processing, and performed a comparative cost assessment with respect to the current state of practice. These results clearly show that this work presents a better, more efficient trade-off between the design-time effort required to develop the initial service line, and the run-time effort required to operate, evolve and maintain the service line as a whole.

A body of knowledge about a computing paradigm typically establishes itself after years of practical application and experience (e.g. the documentation of the GoF design patterns after the wide-spread adoption of object orientation). Cloud computing, and more specifically multi-tenancy at the application level (SaaS), is an increasingly prominent and important software delivery model, which also has a profound impact on how the software is developed and maintained. However, these trends are currently underinvestigated from a software engineering perspective.

The work presented in this paper represents an initial exploration and consolidation of how to develop and build qualitative SaaS applications. More specifically, the presented method focuses on realizing efficiency improvements in terms of reuse and modularity of service variations, as well as on maintenance and evolution aspects of the entire service line.

Acknowledgements

We would like to thank Awais Rashid for the fruitful discussions on the concept of service lines. We also thank the reviewers for their helpful comments to improve this paper. This research is partially funded by the Research Fund KU Leuven (project GOA/14/003 – ADDIS) and by the CUSTOMSS project, which is co-funded by iMinds (Interdisciplinary Institute for Technology), a research institute founded by the Flemish Government. Companies and organizations involved in the project are Agfa Healthcare, Televic Healthcare and UnifiedPost, with project support of IWT.

References

Abu-Matar, M., Gomaa, H., 2011. Variability modeling for service oriented product line architectures. In: SPLC '11: 15th International Software Product Line Conference, pp. 110–119.

Bachmann, F., Goedicke, M., Leite, J., Nord, R., Pohl, K., Ramesh, B., Vilbig, A.,2004. A meta-model for representing variability in product family development. In: PFE '04: Software Product-Family Engineering. Springer Berlin/Heidelberg, pp. 66–80.

Baresi, L., Guinea, S., Pasquale, L., 2012. Service-oriented dynamic software product lines. Computer 45 (10), 42–48.

Bass, L., Clements, P., Kazman, R., 2003. Software Architecture in Practice, 2nd edition. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Bezemer, C.P., Zaidman, A., Platzbeecker, B., Hurkmans, T., Hart, A., 2010. Enabling multi-tenancy: an industrial experience report. In: ICSM '10: 26th International Conference on Software Maintenance, pp. 1–8.

Cavalcanti, Y.C., do Carmo Machado, I., da Mota, P.A., Neto, S., Lobato, L.L., de Almeida, E.S., de Lemos Meira, S.R.,2011. Towards metamodel support for variability and traceability in software product lines. In: VaMoS '11: 5th Workshop on Variability Modeling of Software-Intensive Systems. ACM, pp. 49–57.

Chong, F., Carraro, G., April 2006. Architecture Strategies for Catching the Long Tail. Microsoft Corporation http://msdn.microsoft.com/en-us/library/aa479069.aspx

Classen, A., Boucher, Q., Heymans, P., 2011. A text-based approach to feature modelling: syntax and semantics of TVL. Science of Computer Programming 76 (12), 1130–1143.

Clements, P., Northrop, L.M., 2001. Software Product Lines: Practices and Patterns. Addison-Wesley.

Cohen, S., Krut, R.,May 2008. Proceedings of the First Workshop on Service-Oriented Architectures and Software Product Lines. Carnegie Mellon University – Software Engineering Institute.

CUSTOMSS, 2011. CUSTOMization of Software Services in the cloud (iMinds ICON project). http://www.iminds.be/en/research/overview-projects/p/detail/customss

Czarnecki, K., Helsen, S., Eisenecker, U., 2005. Staged configuration through specialization and multilevel configuration of feature models. Software Process: Improvement and Practice 10 (2), 143–169.

Dumitraş, T., Narasimhan, P.,2009. Why do upgrades fail and what can we do about it? In: Middleware '09: 10th ACM/IFIP/USENIX International Conference on Middleware. Springer Berlin/Heidelberg, pp. 349–372.

Etien, A., Salinesi, C.,2005. Managing requirements in a co-evolution context. In: 13th IEEE International Conference on Requirements Engineering. IEEE, pp. 125–134.

Fowler, M., January 2004. Inversion of Control Containers and the Dependency Injection pattern. http://martinfowler.com/articles/injection.html

Galster, M.,2010. Describing variability in service-oriented software product lines. In: ECSA '10: 4th European Conference on Software Architecture: Companion Volume. ACM, pp. 344–350.

Galster, M., Avgeriou, P., 2012. A variability viewpoint for enterprise software systems. In: Joint Working IEEE/IFIP Conference on Software Architecture (WICSA) and European Conference on Software Architecture (ECSA), pp. 267–271.

Gey, F., Walraven, S., Landuyt, D., Joosen, W.,2013. Building a customizable Business-Process-as-a-Service application with current state-of-practice. In: SC '13: 12th International Conference on Software Composition. Springer Berlin/Heidelberg, pp. 113–127.

Groher, I., Weinreich, R., 2012. Integrating variability management and software architecture. In: Joint Working IEEE/IFIP Conference on Software Architecture (WICSA) and European Conference on Software Architecture (ECSA), pp. 262–266.

Günther, S., Berger, T., 2008. Service-oriented product lines: towards a development process and feature management model for web services. In: SOAPL '08: Workshop on Service-Oriented Architectures and Software Product Lines, pp. 131–136.

Guo, C.J., Sun, W., Huang, Y., Wang, Z.H., Gao, B., July 2007. A framework for native multi-tenancy application development and management. In: CEC/EEE '07: 9th IEEE International Conference on E-Commerce Technology and 4th IEEE International Conference on Enterprise Computing, E-Commerce, and E-Services, pp. 551–558.

Hallsteinsen, S., Hinchey, M., Park, S., Schmid, K., 2008. Dynamic software product lines. Computer 41 (4), 93–95.

Hallsteinsen, S., Jiang, S., Sanders, R., 2009. Dynamic software product lines in service oriented computing. In: DSPL '09: 3rd International Workshop on Dynamic Software Product Lines, pp. 28–34.
Haugen, Ø., et al., 2012. Common Variability Language (CVL). http://www.omgwiki.org/variability/
Hubaux, A., Heymans, P., Schobbens, P.-Y., Deridder, D., Abbasi, E., 2011. Supporting multiple perspectives in feature-based configuration. Software and Systems Modeling, 1–23.
Kabbedijk, J., Jansen, S., 2011. Variability in multi-tenant environments: architectural design patterns from industry. In: Advances in Conceptual Modeling. Recent Developments and New Directions. Springer Berlin/Heidelberg, pp. 151–160.
Kabbedijk, J., Salfischberger, T., Jansen, S., 2013. Comparing two architectural patterns for dynamically adapting functionality in online software products. In: PATTERNS '13: 5th International Conferences on Pervasive Patterns and Applications, pp. 20–25.
Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S., 1990. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Tech. Rep. 21. Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA.
Kang, K.C., Kim, S., Lee, J., Kim, K., Shin, E., Huh, M., 1998. FORM: a feature-oriented reuse method with domain-specific reference architectures. Annals of Software Engineering 5 (1), 143–168.
Kastner, C., Thum, T., Saake, G., Feigenspan, J., Leich, T., Wielgorz, F., Apel, S., May 2009. FeatureIDE: a tool framework for feature-oriented software development. In: ICSE '09: 31st IEEE International Conference on Software Engineering, pp. 611–614.
Kotonya, G., Lee, J., Robinson, D., 2009. A consumer-centred approach for service-oriented product line development. In: WICSA/ECSA '09: Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture, pp. 211–220.
Kruchten, P.B., 1995. The 4+1 view model of architecture. IEEE Software 12 (6), 42–50.
Lee, J., Kotonya, G., 2010. Combining service-orientation with product line engineering. IEEE Software 27 (3), 35–41.
Medeiros, F.M., de Almeida, E.S., de Lemos Meira, S.R., 2009. Towards an approach for service-oriented product line architectures. In: SOAPL '09: 3rd Workshop on Service-Oriented Architectures and Software Product Lines, pp. 151–164.
Mendonca, M., Branco, M., Cowan, D.,2009. S.P.L.O.T.: software product lines online tools. In: OOPSLA '09: 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications. ACM, pp. 761–762.
Mietzner, R., Leymann, F., 2008. Generation of BPEL customization processes for SaaS applications from variability descriptors. In: SCC '08: IEEE International Conference on Services Computing, pp. 359–366.
Mietzner, R., Leymann, F., Unger, T., 2011. Horizontal and vertical combination of multi-tenancy patterns in service-oriented applications. Enterprise Information Systems 5 (1), 59–77.
Mietzner, R., Metzger, A., Leymann, F., Pohl, K.,2009. Variability modeling to support customization and deployment of multi-tenant-aware Software as a Service applications. In: PESOS '09: ICSE Workshop on Principles of Engineering Service Oriented Systems. IEEE Computer Society, pp. 18-25.
Nguyen, T., Colman, A., Han, J., 2012. Enabling the delivery of customizable web services. In: ICWS '12: 19th IEEE International Conference on Web Services, pp. 138–145.
Nguyen, T., Colman, A., Talib, M.A., Han, J.,2011. Managing service variability: state of the art and open issues. In: VaMoS '11: 5th Workshop on Variability Modeling of Software-Intensive Systems. ACM, pp. 165–173.
Nuseibeh, B., 2001. Weaving together requirements and architectures. Computer 34 (3), 115–119.
Parra, C., Blanc, X., Duchien, L., 2009. Context awareness for dynamic service-oriented product lines. In: SPLC '09: 13th International Software Product Line Conference, pp. 131–140.
Pawlak, R., Seinturier, L., Duchien, L., Florin, G.,2001. JAC: a flexible solution for aspect-oriented programming in Java. In: REFLECTION '01: 3rd International Conference on Metalevel Architectures and Separation of Crosscutting Concerns. Springer-Verlag, pp. 1–24.
Pohl, K., Böckle, G., Van Der Linden, F., 2005. Software Product Line Engineering: Foundations, Principles, and Techniques. Springer-Verlag, New York.
Popovici, A., Alonso, G., Gross, T.,2003. Just-in-time aspects: efficient dynamic weaving for Java. In: AOSD '03: 2nd International Conference on Aspect-oriented Software Development. ACM, pp. 100–109.
Schroeter, J., Cech, S., Götz, S., Wilke, C., Aßmann, U.,2012a. Towards modeling a variable architecture for multi-tenant SaaS-applications. In: VaMoS '12: 6th International Workshop on Variability Modeling of Software-Intensive Systems. ACM, pp. 111–120.
Schroeter, J., Mucha, P., Muth, M., Jugel, K., Lochau, M.,2012b. Dynamic configuration management of cloud-based applications. In: SPLC '12: 16th International Software Product Line Conference – Volume 2. ACM, pp. 171–178.
SpringSource, 2013. Aspect-oriented Programming with Spring. http://static.springsource.org/spring/docs/4.0.x/spring-framework-reference/html/aop.html
Sun, W., Zhang, X., Guo, C.J., Sun, P., Su, H., 2008 Sept. Software as a Service: configuration and customization perspectives. In: SERVICES-2 '08: IEEE Congress on Services Part II, pp. 18–25.
Truyen, E., Vanhaute, B., Jorgensen, B.N., Joosen, W., Verbaeten, P.,2001. Dynamic and selective combination of extensions in component-based applications. In: ICSE '01: 23rd International Conference on Software Engineering. IEEE Computer Society, pp. 233–242.
van der Storm, T.,2007. Generic feature-based software composition. In: SC '07: International Conference on Software Composition. Springer Berlin/Heidelberg, pp. 66–80.
van Deursen, A., Klint, P., 2002. Domain-specific language design requires feature descriptions. Journal of Computing and Information Technology 10 (1), 1–17.
Van Landuyt, D., Op de beeck, S., Truyen, E., Joosen, W.,2009. Domain-driven discovery of stable abstractions for pointcut interfaces. In: AOSD '09: 8th ACM International Conference on Aspect-oriented Software Development. ACM, pp. 75–86.
Walraven, S., Truyen, E., Joosen, W.,2011. A middleware layer for flexible and cost-efficient multi-tenant applications. In: Middleware '11: Proceedings of the 12th ACM/IFIP/USENIX International Conference on Middleware. Springer Berlin/Heidelberg, pp. 370–389.
Walraven, S., Van Landuyt, D., Gey, F., Joosen, W., November 2013. Service line engineering in practice: Developing an integrated document processing SaaS application. CW Reports 652. Department of Computer Science, KU Leuven http://www.cs.kuleuven.be/publicaties/rapporten/cw/CW652.abs.html

Stefan Walraven is a PhD candidate in the Department of Computer Science at KU Leuven, Belgium, and a member of the research group iMinds-DistriNet. He received a Masters degree in Computer Science from KU Leuven in 2008. His main research interests include adaptive middleware and software engineering, more specifically in the context of multi-tenant cloud applications.

Dimitri Van Landuyt is a postdoctoral researcher at iMinds-DistriNet, the Distributed Systems research group of the Department of Computer Science of KU Leuven, Belgium. Dimitri obtained a PhD in Computer Science in 2011 and focuses his current research efforts on applying and validating established software engineering principles to cloud computing, more specifically in the context of Software-as-a-Service applications.

Eddy Truyen received a PhD degree in Computer Science from KU Leuven, Belgium in 2004. In 2004–2009, he was a postdoctoral researcher at the Department of Computer Science of KU Leuven and member of the research group iMinds-DistriNet. In 2009, he obtained a permanent position at iMinds-DistriNet as Research expert. His main research activities are in the area of software engineering, adaptive middleware, dynamic re-configuration and engineering of customizable software services. He has been involved in national and international projects on adaptive middleware and cloud computing.

Koen Handekyn is CEO of UP-nxt, the R&D entity within the UnifiedPost group, and also holds the position of CTO within the group. Koen holds a degree in Computer Science engineering from UGent, and a degree in Business Administraton of Vlerick Leuven Gent Management School. He has a background in Telecommunications and Innovation, and nurtures a lifelong interest in software engineering, loving sharing his experiences. He survives the increasing traffic by embracing – with the team – all technologies that help for teleworking.

Wouter Joosen is full professor in distributed software systems at the Department of Computer Science of KU Leuven, Belgium. He obtained a PhD degree from KU Leuven in 1996. He has also co-founded spin-off companies of KU Leuven: Luciad, a company specializing in software components for Geographical Information Systems, and Ubizen (now part of Verizon Business Solutions), where he has been the CTO from 1996 till 2000, and COO from 2000 till 2002. His current research interests are in distributed systems and cloud computing, focusing on software architecture and adaptive middleware, as well as in security aspects of software.