Feature Models at Run Time

Feature Middleware for Multi-tenant SaaS Applications

Fatih Gey, Dimitri Van Landuyt, Stefan Walraven and Wouter Joosen

iMinds-DistriNet, KU Leuven, Celestijnenlaan 200A, 3001 Leuven, Belgium
{firstname}.{lastname}@cs.kuleuven.be

Abstract. Software product line engineering (SPLE) techniques revolve around a central variability model which in many cases is a feature model that documents the logical capabilities of the system as features and the variability relationships between them. In more traditional SPLE, this feature model is a result of domain analysis and requirement elicitation, while more recently this approach has been extended to represent also design-time variability, for example to document different ways to realize the same functionality.
In many approaches, the feature model has run-time relevance as well. For example, in earlier work, we have used SPLE techniques to develop customizable multi-tenant SaaS applications, i.e. SaaS applications of which a single run-time instance is offered to many customer organizations (tenants), often with widely different requirements. In such systems, tenant customization is accomplished entirely at run time.
In this paper, we present and explore the idea of promoting the feature model as a run-time artifact in the context of customizable multi-tenant SaaS applications, and we discuss the potential benefits in terms of the deployment, operation, maintenance, and evolution of these systems. In addition, we discuss the requirements this will impose on the development methods, the variability modeling languages, and the middleware.

1 Introduction

Software as a Service (SaaS) is a software delivery model in which the software is offered as an online service by a SaaS provider and remotely accessed by different customer organisations to which we refer hereafter as tenants. In order to optimize cost efficiency, SaaS providers aim to exploit economies-of-scale effects by sharing resources and artifacts among a large number of tenants; when sharing even run-time instances, it is called a multi-tenant SaaS application [1]. As different tenants impose different requirements, a multi-tenant SaaS application typically supports a number of variations, which affects the design of the application profoundly.

Software Product Line Engineering (SPLE) [2] introduces software development techniques that focus on structuring a set of closely-related application variants and supporting application-level variability for the early stages of development. A SPL revolves around a central variability model (i.e. in many cases

a feature model [3, 4]) that documents the logical capabilities of the system as features and the variability relationships between them.

In earlier work [5], we have applied SPLE techniques to systematically prepare the multi-tenant SaaS application for this type of run-time variability. Specifically, we have presented a methodology called *Service Line Engineering* (SLE). As with traditional SPLE [2], the feature model is a key artifact that defines variability supported by the MT SaaS application at run time. Variability is realized at the architecture level by defining a number of components which are interchangeable and correspond to different features defined in the feature model. More specifically, feature-to-composition mappings are created that document the traceability link between a feature and its realization in terms of technical compositions (between beans, packages, classes, etc.) and that can be activated at run time. Maintaining these traceability relations, ensuring completeness (e.g. for every feature there must be at least one realization) is a manual and complex development task, and these mappings are prone to architectural erosion. For example, in a prototype [6], we have realized variability in the context of a SOA system by supporting customization at the level of the overall business process workflow and the individual back-end services. A key difference however with traditional and more recent applications of SPLE is that activation of variants is not done at design, packaging, or deploy time, but at run time: based on a specific incoming tenant request, the service line will activate different application configurations that comply to the tenants requirements. As for many other, in our case, variability support is based on custom development of common requirements, which is an error-prone task that is better provided by a middleware.

In this paper, we argue that in the context of such configurable multi-tenant SaaS applications a feature is a useful abstraction to leverage from development to run time. We present our vision on a *Feature Middleware*, i.e. middleware that supports feature decomposition as the main component model and run-time activation of different feature configurations, and we outline our research roadmap to realize this vision.

The remainder of this paper is structured as follows: Section 2 presents the potential role of features at run time in the context of customizable multi-tenant SaaS applications. Section 3 discusses the gap analysis w.r.t. the current state-of-the-art (Sec. 4) and outlines our roadmap in terms of realizing this more advanced feature middleware. Finally, Section 5 concludes the paper.

2 Motivation and Benefits

In the current state of the art of SPLE-based techniques that support customization at run time (e.g. [7, 8, 5, 9]), the feature itself is a development abstraction, compiled away and invisible at run time. This section argues why features would be suitable abstractions also beyond the development stages, i.e. for structuring the run time of, for operating, maintaining and evolving a customizable multi-tenant SaaS application. This is based on the observation that many key

activities rely on translating lower-level abstractions into features and vice versa, and that this translation has to be done by many different stakeholders involved in the eco-system of the multi-tenant SaaS application.

2.1 Stakeholder-specific Requirements at Run Time

The *SaaS developer* is concerned with the technical realization of the variability expressed in the feature model. Creating a system that has the capabilities to dynamically adapt itself at the basis of specific tenant requests is a complex endeavour that relies heavily on middleware support. In our vision of a feature middleware, variability is supported explicitly at the level of features instead of lower layer, technical compositions. When adding or changing a feature, developers could additionally validate planned changes against the currently used feature model. Moreover, the set of impacted tenants could be determined from the tenant's feature selections using the run-time feature model.

The *tenant administrator* is an employee of the tenant organisation, responsible for configuring the application to the requirements of the tenant organisation. Typically, configuration interfaces either expose internals and low-level details, or are custom developed to provide application abstractions and thus are costly to co-evolve with the SaaS offering. Exposing the feature model as a configuration interface is a good compromise to allow the tenant some insight into the SaaS offering and enable some degree of self-service [10]. Self-service is a key enabler for scalability and profitability of SaaS offerings: by allowing the tenant to register himself, no extra cost or effort is required from the SaaS provider, and economies-of-scale benefits can be obtained. Self-service remains beneficial beyond the initial setup, especially when complemented with monitoring data: By reporting about usage and ((un-)expected) application events, the system would leverage self-service for maintenance and debugging, e.g. tenant administrators would be able to reconfigure their feature selection based on their actual cost, or wish to disable features that cause the application to malfunction.

Similar benefits of transparency applies for the *billing department* of the SaaS provider. Typically, billing is done in terms of either resource usage of the underlying platform (CPU hours, memory usage, etc.), or in terms of application-level abstractions (service hours). While the latter case is more natural, in many cases billing support for these application abstractions typically have to be custom-developed by the SaaS developer. Billing at the basis of individual features (feature hours) will enable (i) more fine-grained pricing schemes and policies (e.g. discounts at the basis of feature combinations, premium versus regular service levels, etc.), and (ii) the creation of reusable and generic middleware support of such complex billing functionalities (policy-based billing middleware support).

After the initial design and implementation of the Multi-tenant SaaS application, the focus shifts away from these initial development stages to run-time operation and evolution. Allowing the inspection of a run-time SaaS application in terms of features will enable the SaaS operator to map the impact of a technical change to affected features, and furthermore to affected tenants: since multiple tenants are serviced by one operational instance, these systems

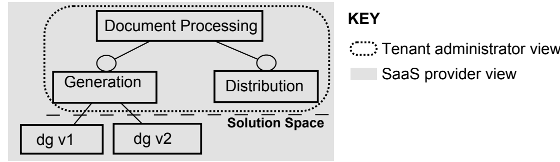

Fig. 1. Document Processing Application using Features at Run Time (Feature Model inspired by FODA [3]).

should be adaptable without any service disruption. The feature as an explicit abstraction at run time could be a major enabler for a gradual, fine-grained and impact-sensitive evolution: updates could be partially applied to avoid impact on on-going operations, in the sense that updated and previous versions of feature are operated in parallel, and older versions are gradually removed.

2.2 Illustration

This section presents our vision of features at run time by means of an example application for document processing. This multi-tenant SaaS application offers services for generating, signing, distributing and archiving digital documents to B2B customers (cf. [6]). For illustration purposes, we limit the set of features to two optional ones: `document generation` and `document distribution` (see Figure 1). The presented feature model indicates that `document generation` is deployed in two versions (`dg v1` and `dg v2`), which refers to solution space variability that is typically not modelled together with problem space elements.

By *having* these different types of variability in the same model, a SaaS provider can reason about the implementation artifacts `dg v1` and `dg v2` in terms of the higher-level abstraction `Document Generation` (e.g. when troubleshooting). At the same time, for example, a tenant administrator can remain agnostic of `dg v1` and `dg v2`, e.g. when inspecting the costs of `Document Generation` on a dashboard.

3 Requirements and Challenges

In this section, we discuss the features at run time from an engineering methodology perspective, point-out limitations of current feature modelling techniques, and present our vision on a middleware support for features at run time, while referring to related work.

3.1 Engineering Methodology

Ideally, we would like to think of a multi-tenant SaaS application as a collection of features: implement and test features during the development independently, configure and compose them to a meaningful application at run time, and add

evolved features to the application after its initial release. This imposes the following requirements to the engineering methodology:

R1. Modular development of a feature, i.e. independent of other features
R2. Production of reusable and composable features
R3. Features are automatically composable at run time

Aspect-oriented Software Development [11] uses Aspects (AOP [12]) to employ modular development of a concern that may affect multiple program locations. It is highly related to R1, as features are likely to affect other features' implementations.

Feature-oriented Software Development (FOSD) [7] is an engineering concept that relates even closer to our vision and requirements, as it shares our requirements to combine the aspects of modular development with reusable features as resulting artifacts: To realize the former, the methodology relies on feature-oriented programming [13] in which a program is modelled as layers of *feature artifacts*, that is, the code of a feature artifact is applied to a base program. In that, it is similar to AOP and often uses aspect-oriented techniques for deployment [7]. To realize the latter, FOSD uses feature modelling as known from SPLE to structure the requirements into common and variable items. In contrast to FOSD, we are aiming to use components as the unit of modules that implements a feature, as components in their nature (i.e. through encapsulation and information hiding) offer better reusability.

However, we share two challenges with FOSD when it comes to composition. First, composing feature implementations is non-trivial due to *feature interactions* [14] (R2). Second, in FOSD and multi-tenant SaaS applications that provide configuration through a self-service interface, automated instantiation of the application is highly desired. Yet, requirements-describing features and variant implementations usually map in a *many-to-many relationship* [15], but producing a variant composition from a given selection of features requires a unique mapping from requirements and implementation artifacts (R3); this transformation is non-trivial.

3.2 Dynamic and Decentralized Feature Model

Our motivation to compose at run time stems from the goal to specialize implementation artifacts (towards tenant specificities) the latest possible, in order to maximize the potential for sharing it at run time among other tenants. Van Gurp et al. have observed that in order to maintain variability throughout the development phases, "design decisions are delayed and left open for variability deliberately" [15]. This results in a more complex feature model.

In addition, a SaaS application is subject to different types of variability at run time. One type results from frequently changing the context-based trade-off setting between cost, performance, latency, and other properties during operation and on a per-tenant basis: a SaaS application may choose to dispatch between different *deployments* of the same implementation, or between different

implementations (e.g. from different vendors, as for databases). When evolving, updated versions of run-time components are typically operated in parallel to ensure service continuity while phasing out previous versions gradually. In such a case, the *co-existence of multiple versions* of the same component depicts another type of variability. Being a service among others, a SaaS application may also vary in implementing a certain feature with *in-house or external* services. This list of run-time-related types of variability is not meant to be exhaustive, but to provide a coarse-gained impression of different types of variability. As motivated earlier, there is a significant benefit in fully maintaining the compositional model (which includes all kinds of variability necessary for a multi-tenant SaaS application) as an implementation artifact during development and run time.

Traditional feature modelling techniques, however, focus on modelling requirements (thereby not distinguishing between different types of variability), and typically create a model first and translate it afterwards into an architecture and implementation [16]. Modelling the additional complexity that features at run time expose – additional complexity that results from the delayed design decisions during development, and from necessary flexibilities at run time – would render the feature model impractical. The same complexity makes also frequent changes likely, and thereby such a complex feature model fragile. As a result, we expect the following requirements from a suitable feature model for multi-tenant SaaS applications:

R1. Native support of *different types of variability*
R2. To be *decentralized* in a way that meta-data that describes a single feature is stored close to its application context, and a feature model is dynamically constructed from a set distributed feature descriptions

Next, we reason how those requirements are beneficial to tackle said gaps. One benefit is the capability to automatically provision views that do and do not include a certain type of variability (such as versions) to tenant administrators and SaaS provider. Another type, for example: deployment-related variability could be entirely hidden from operators and administrators, but be subject to a self-adaptive control loop. The most prevalent benefit of a dynamic and decentralized feature model is *access to a consistent feature model* at run time. More specifically, a feature model that is constructed from artifacts of a running application at run time has the benefit over a design-time artifact that it cannot be outdated and always provides a consistent model of the application. In addition, it naturally provides a *smaller scope for reading, processing or changing a feature model*. For example, SaaS developers that are interested in learning about *only a certain part* of the application (e.g. for extending or debugging) could dynamically construct only the feature model that is related to the features of their interest. Moreover, processing the feature model (e.g. for analysis) can naturally be limited in scope as in the previous example, from which *processing time and complexity* benefits significantly. When evolving the feature model, a dynamic and decentralized approach facilitates to *limit the impact on the adaptation*, as changes are localized and components that dynamically fetch all feature descrip-

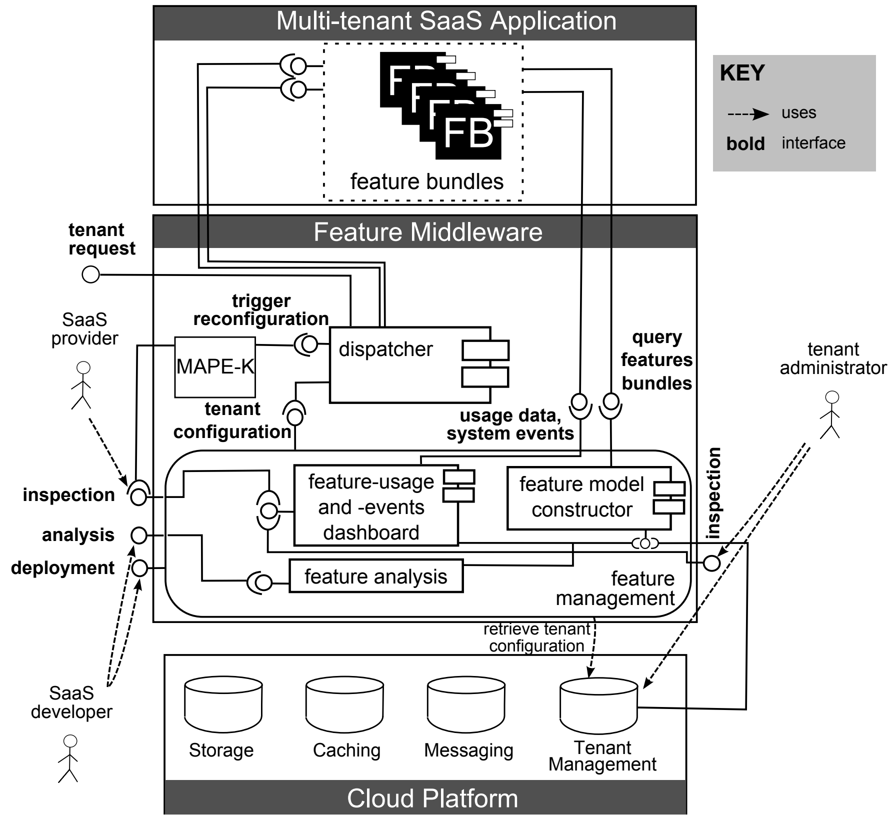

Fig. 2. Feature Middleware: Execution Environment and Services for Feature Bundles.

tions to construct a feature model will naturally adapt at the next time at which they retrieve information about the changed feature.

3.3 Feature Middleware

The composition model we envision consists of components that have a descriptive, modelling part (including interrelations and constraints) and an implementation part of features. We refer to both parts together as a *feature bundle*. Figure 2 shows our vision of a middleware that operates on the basis of feature bundles, which will be elaborated next. Note that certain services are assumed from the underlying cloud platform; for the sake of scoping the presentation in this paper to the Feature Middleware, a tenant management service is included that provides management of tenant accounts and provides a self-service interface for the tenant administrator to setup and configure the SaaS application.

Each service request from a tenant application is received from the `dispatcher` component that queries the tenant configuration via the `feature management` component from the cloud platform, and prepares the execution context for that

request. If necessary, this may include to spawn additional instances of feature bundles. Thereafter, the tenant configuration is added to the request which is then dispatched to the according feature bundles.

At any time during operation, the tenant administrator (TA) can use an `inspection interface` to access a dashboard that presents usage history as well as the current load of its configured features, and thereby essentially the current consumption and cost of those features. This enables the TA to adjust the configuration and to replace features that are not beneficial against other features autonomously. The same dashboard also captures and presents events of the TA's features. This facilitates self-service for debugging (cf. Section 2).

The same capabilities are available for the SaaS provider but without a view that is limited to a tenant context. This dashboard could additionally be used as an input for a self-adaption control loop (cf. `MAPE-K` component).

In order to present this information at the abstraction of features, the component `feature model constructor` observes deployed feature bundles and actively maintains a constructed feature model, while the `feature-usage and -events dashboard` component logs single events and presents them in different (e.g. accumulated) views.

The Feature Middleware provides also interfaces for the SaaS provider, if not the most significant ones. Using the `analysis` interface, he can understand the impact of updating a feature bundle. A list of affected tenants can be queried, given a set of feature bundles that are potentially subject to change. Additionally, planned changes can be simulated on the constructed feature model in order to validate the results or to perform consecutive simulations. Finally, by using the `deploy` interface, a SaaS developer can safely deploy a new feature bundle. That is, only updates that are supposed not to impact the correctness of any on-going operation, will replace active features, others are deployed but not activated and will require manual activation via the tenant configuration interface.

4 Related Work

Dynamic SPLE (DSPLE) [8] is a very prominent SPLE-based approach that supports variability activation at run time. Typically, development models [17] that express variability are compiled into run-time models [17] that facilitate adaptation of the application at run time using solution-space abstractions (cf. Sec. 2).

In DSPLE, adaptation at run time is often understood (e.g. [18, 19]) as a necessary response of the application to changes in the computing environment (e.g. the underpinning (cloud) infrastructure) in order to meet or to maintain functional or non-functional goals. Some of these approaches (e.g. [20–22]) additionally support the evolution of variability at run time. For example, Morin et al. [22] propose a separate adaptation management [23] to control the main application's adaptation using a goal-driven approach, and which can be evolved (for example, by changing the goals) at run time.

These approaches require the necessary run-time models to be up-to-date and thus manually synchronized with development models. In our target domain –

large-scale multi-tenant SaaS applications with a numerous amount of differently typed variations – such an assumption seems impractical. By merging development and run-time models, our envisioned solution aims at naturally providing a variability model that is constructed at run time and is thus at all times up-to-date. Moreover, it provides stakeholder-friendly problem-space abstractions to facilitate self-management at run time. In addition, we envision the feature model to be decomposed and distributed to facilitate evolution (to accommodate changing application requirements) with minimal impact through dynamic adaptation.

5 Conclusion and Future Work

Efficiently managing and operating a SaaS application that provides variability at large scale is challenging. In ongoing research, we aim at efficient development and operation of customizable multi-tenant SaaS applications. Specifically in this paper, we motivated feature decomposition as main component model at development and run time. We pointed out the benefits of modular development for a SaaS developer, increased opportunities for self-service (inducing lower operations costs) and decreased impact of evolution on service continuity for the SaaS operator. We discussed the implied complexity that comes with such a component model and assessed that current feature modelling techniques lack the support for additional types of variability that occur at run time. Finally, we presented our vision on a middleware that tackles this complexity in an efficient way.

This paper presents our mid-term research goals, in the sense that in future work, we will address the challenges sketched in Section 3, and will validate the resulting middleware in the context of realistic SaaS offerings [24, 25].

Acknowledgments. This research is partially funded by the Research Fund KU Leuven and by the projects ADDIS, DMS^2 and D-BASE. The projects DMS^2 and D-BASE are projects co-funded by iMinds (Interdisciplinary Institute for Technology) a research institute founded by the Flemish Government.

References

1. Chong, F., Carraro, G.: Architectural strategies for catching the long tail., `http://msdn.microsoft.com/en-us/library/aa479069.aspx`. MSDN Website (2006)
2. Pohl, K., Böckle, G., Van Der Linden, F.: Software product line engineering: foundations, principles, and techniques. Springer (2005)
3. Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, A.: Feature-oriented domain analysis (FODA) feasibility study. Technical report, DTIC Document (1990)
4. Kang, K.C., Lee, J., Donohoe, P.: Feature-oriented product line engineering. IEEE Softw. **19** (2002) 58–65
5. Walraven, S., Van Landuyt, D., Truyen, E., Handekyn, K., Joosen, W.: Efficient customization of multi-tenant software-as-a-service applications with service lines. Journal of Systems and Software **91** (2014) 48–62

6. Walraven, S., Van Landuyt, D., Gey, F., Joosen, W.: Service line engineering in practice: Developing an integrated document processing saas application. CW Reports CW652, Department of Computer Science, KU Leuven (2014)
7. Apel, S., Kästner, C.: An overview of feature-oriented software development. Journal of Object Technology **8.5** (2009) 49–84
8. Hallsteinsen, S., Hinchey, M., Park, S., Schmid, K.: Dynamic software product lines. Computer **41** (2008) 93–95
9. Baresi, L., Guinea, S., Pasquale, L.: Service-oriented dynamic software product lines. Computer **45** (2012) 42–48
10. Sun, W., Zhang, X., Guo, C.J., Sun, P., Su, H.: Software as a service: Configuration and customization perspectives. In: Proceedings Services-2, IEEE (2008)
11. Filman, R., Elrad, T., Clarke, S., Ak?it, M.: Aspect-oriented Software Development. First edn. Addison-Wesley Professional (2004)
12. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In: ECOOP. Springer (1997)
13. Prehofer, C.: Feature-oriented programming: A fresh look at objects. In Akit, M., Matsuoka, S., eds.: ECOOP. Springer Berlin Heidelberg (1997)
14. Calder, M., Kolberg, M., Magill, E.H., Reiff-Marganiec, S.: Feature interaction: a critical review and considered forecast. Computer Networks **41** (2003)
15. Van Gurp, J., Bosch, J., Svahnberg, M.: On the notion of variability in software product lines. In: Proceedings Software Architecture. (2001)
16. Czarnecki, K., Grünbacher, P., Rabiser, R., Schmid, K., Wasowski, A.: Cool features and tough decisions: A comparison of variability modeling approaches. In: Proceedings VaMoS, ACM (2012)
17. France, R., Rumpe, B.: Model-driven development of complex software: A research roadmap. In: Proceedings Future of Software Engineering (FOSE), IEEE (2007)
18. Kumara, I., Han, J., Colman, A., Kapuruge, M.: Runtime evolution of service-based multi-tenant saas applications. In: Proceedings ICSOC. Springer (2013)
19. Almeida, A., Cavalcante, E., Batista, T., Cacho, N., Lopes, F., Delicato, F., Pires, P.: Dynamic adaptation of cloud computing applications. In: International Conference on Software Engineering and Knowledge Engineering. (2013)
20. Chauvel, F., Ferry, N., Morin, B., Rossini, A., Solberg, A.: Models@runtime to support the iterative and continuous design of autonomic reasoners. In: Proceedings Models@Run.time. (2013)
21. Pasquale, L., Baresi, L., Nuseibeh, B.: Towards adaptive systems through requirements@ runtime. In: 6th Workshop on Models@run.time. (2011)
22. Morin, B., Barais, O., Jezequel, J., Fleurey, F., Solberg, A.: Models@run.time to support dynamic adaptation. Computer **42** (2009) 44–51
23. Oreizy, P., Gorlick, M.M., Taylor, R.N., Heimbigner, D., Johnson, G., Medvidovic, N., Quilici, A., Rosenblum, D.S., Wolf, A.L.: An architecture-based approach to self-adaptive software. IEEE Intelligent Systems **14** (1999) 54–62
24. iMinds: D-BASE Project: Optimization of Business Process Outsourcing Services, `http://www.iminds.be/en/projects/2014/05/06/d-base`. Website (2014)
25. iMinds: DMS2 Project: Decentralized Data Management and Migration of SaaS, `http://www.iminds.be/en/projects/2014/03/06/dms2`. Website (2014)