Feature-Based Application Development and Management of Multi-Tenant Applications in Clouds

Hendrik Moens and Filip De Turck
Ghent University – iMinds
Department of Information Technology
Gaston Crommenlaan 8/201
B-9050 Gent, Belgium
{firstname}.{lastname}@intec.ugent.be

ABSTRACT

In recent years, there has been a rising interest in cloud computing, which is often used to offer Software as a Service (SaaS) over the Internet. SaaS applications can be offered to clients at a lower cost as they are usually multi-tenant: many end users make use of a single application instance, even when they are from different organisations. It is difficult to offer highly customizable SaaS applications that are still multi-tenant, which is why these SaaS applications are often offered in a one size fits all approach.

In some application domains applications must be highly customizable, making it more difficult to migrate them to a cloud environment, and losing the benefits of multi-tenancy. In this paper we compare multiple approaches for the development and management of highly customizable multi-tenant SaaS applications, and present a methodology for developing and managing these applications. We compare two approaches, an application-based approach focusing on deploying multiple multi-tenant applications variants, and a feature-based approach where applications are composed out of multi-tenant services using a service oriented architecture. In addition, we also discuss a hybrid approach combining properties of both. We conclude that the feature-based approach results in the fewest application instances at runtime resulting in more multi-tenancy.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements Specification—*methodologies*; K.6.3 [**Management of Computing and Information Systems**]: Software Management—*software development, software process*; C.2.4 [**Computer Communication Networks**]: Distributed systems—*distributed applications*

Keywords

Cloud computing, feature modeling, multi-tenancy

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
SPLC '14 September 15 - 19 2014, Florence, Italy
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2740-4/14/09 ...$15.00.
http://dx.doi.org/10.1145/2648511.2648519

1. INTRODUCTION

There has been a growing interest in cloud computing, where applications are no longer executed on-premise but in remote datacenters. For application providers, offering Software as a Service (SaaS) via the Internet makes it possible to offer applications to large numbers of end users, resulting in cost savings through economies of scale. These cost-savings are partially achieved by using multi-tenancy: multiple end users make use of the same application instances, reducing the total resource need and reducing the impact of varying numbers of end users. This however results in limited customizability, often causing SaaS applications to offer a one size fits all approach. For some applications, such as e.g. document processing, medical communications and medical information management a very high customizability is needed. In these situations, multiple very large client organizations, referred to as tenants, require extensive customization options for their specific use cases. For these application cases, customizations were traditionally provided on an ad-hoc basis resulting in limited code reuse, increasing management complexity and high development costs. The development, deployment and management approaches discussed in this paper are based on our experiences with these three applications.

Offering multi-tenant SaaS applications with high customizability using a single application instance is hard as all of the customization must then be handled at runtime using a single application binary. Even in cases where this is possible, e.g. if all customizations can be implemented using Aspect-Oriented Programming (AOP) [11], this may still be a problem as executing different code paths for different end users may impact the application performance, and by consequence the quality characteristics of the service. This problem may be further exacerbated by additional quality customizations making it even more difficult to provision the application for all users using a single instance type.

These functional and quality considerations make it infeasible to use a single application instance for all application variants. In this situation, two approaches can be used to offer customizable SaaS applications. 1) Multiple application instances can be built that each include a different set of customizations. In this scenario, multiple multi-tenant application variants are built statically and managed independently. Multi-tenancy can then only be achieved when two tenants make use of the same application customization. We refer to this approach as the Application-Based Binary (ABB) approach. 2) Alternatively, an approach where every appli-

cation is built using a Service-Oriented Architecture (SOA) can be used. We refer to this approach as the Feature-Based Binary (FBB) approach because each of the application services is used to offer one or more of the application features. In this paper, we analyze the approaches for offering customizable multi-tenancy, comparing development, deployment and runtime management; and the amount of multi-tenancy that can be attained using both approaches. Additionally, we propose a hybrid approach that adds properties of the ABB approach to the FBB approach.

In the next section we discuss related work. Afterwards, in Section 3 we discuss the ABB and FBB approaches for managing variability of SaaS applications in clouds and describe a hybrid approach incorporating ABB applications in the FBB approach. Then we analyze the approaches theoretically in Section 4 and experimentally in Section 5. This is followed by a brief discussion in Section 6. Finally, in Section 7 we discuss our conclusions.

2. RELATED WORK

Dynamic software product lines [4, 5] can be used to develop applications that can be customized at runtime. Many different approaches have been proposed in recent years [3]. An aspect-oriented approach for dynamically managing variability is presented in [11]. This approach can be used to create highly customizable applications that all function using a common code base, and thus using common application instances. It is however not always be possible to capture all possible variation using a single code base, limiting the potential customizability of applications. More importantly, using different code paths for different tenants may impact application quality, causing tenants to influence each others performance. Therefore other approaches for offering multi-tenant customizability in SaaS applications, such as the approaches presented in this paper, are still needed. The approaches presented in this paper benefit from such approaches, as they can be used to increase the runtime customizability of individual application features.

Another approach for managing application customizability is by using SOA architectures. An approach using runtime application customization was discussed in [12], where runtime adaptation of applications based on changing application context is achieved for applications built using service-oriented architectures. The authors however do not focus on the running multiple customizations at the same time, which is necessary for multi-tenant applications where tenants have custom customizations.

In [7] Mietzner et al. present an approach for constructing customizable multi-tenant applications using a SOA. This approach is similar to the FBB approach which we discuss in this paper. We however extend the approach by also considering the runtime management and resource allocation of applications. Furthermore, we compare the FBB approach with an alternative ABB approach to determine when each approach is preferable using a theoretical and experimental analysis, and we present a hybrid approach combining properties of both FBB and ABB. In [6], the authors make a distinction between external and internal variation. Only the former variations visible to end users while the other variants may be left undecided when applications are specified resulting in open variation points. We exploit this concept of open variation points to reduce resource costs in the FBB management algorithms.

Some approaches, such as [2], focus on customizing applications by changing the workflow in SOA applications. The FBB approach described in this paper is similar in that we use a SOA, but we focus on replacing components based on tenant customizations for performance isolation rather than on customizing the interactions between the components. These workflow customization approaches are complementary to the FBB approach as they can be used to coordinate between the resulting application components and offer additional application customizability.

The concepts presented in this paper are based on service lines [16], which are used to construct customizable workflows of customizable services. The authors however focus on AOP and dependency injection to offer customizations. The FBB approach presented in this paper extends the approach by offering greater customizability of services by permitting the use of separate application instances when doing so is needed for performance isolation and higher customizability. Furthermore, the management approach discussed in this paper can be used to reduce management costs by exploiting open variation points at runtime.

This paper builds on our previous work in runtime management of customizable cloud applications [9, 10], where we discussed how SOA applications can be managed by a cloud management system, and how open variation points can be exploited at runtime to reduce management costs. This paper describes the broader approach, discussing how these component-based applications can be developed, deployed and managed. We also compare the FBB approach with an alternative ABB approach, and describe a hybrid approach containing properties of both approaches.

3. SaaS MULTI-TENANCY APPROACHES

Multi-tenancy is an important concept for reducing costs in cloud environments. When an application is multi-tenant, multiple end users and tenants make use of a single shared application instance. An *instance* in this context is a compiled artifact that is executed on physical or virtualized hardware. The different ways in which variability is handled can have an impact on development complexity, performance consistency and flexibility. We focus on an approach using feature modeling, where the variability of an application is represented using *features*. A feature is a specific application functionality that can be included in an application. These functionalities can be represented in a feature model, a formal representation of relations between features that can be used to determine which feature combinations are possible. We consider two types of variability:

1. **Compile time variation:** If variations are compiled into the application, this results in the maximum flexibility when developing applications as entirely different code may be used for each different application variant. Furthermore, it is easier to ensure there is more consistent application performance for different users as all users make use of the same code. The cost of this approach is however higher, as it is impossible to share resources between tenants with different configurations. This results in an increase of instance types and management complexity. These variations can be either defined by developing separate code modules, or alternatively by defining AOP aspects that are compiled statically into the application binary.

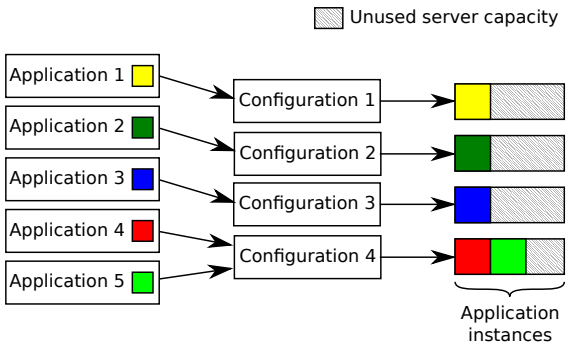
2. **Runtime variation:** Two runtime variation types can be distinguished: configuration changes and customization changes [13]. Runtime configuration can only be used for smaller changes that can be done by e.g. changing configuration files. These changes are easy to implement and are therefore already supported by many SaaS applications. They are however also the least flexible. Runtime customization changes are harder to implement as they result in the execution of different code paths. Using AOP techniques, the code of running applications can be changed at runtime by dynamically weaving changes into the runtime binary of an application. While dynamic AOP is well-established, it is more complicated to develop and test applications using this approach compared to an approach where static compilation is used.

In practice applications have multiple customization and configuration options. Therefore, it is possible to combine the various variation generation approaches, handling some of the changes at runtime while managing others at compile time. As in pure multi-tenant applications all end users make use of the same application instance, customization can only be offered through runtime variation. This limits the customizability of the application. Alternatively, an approach with multiple application instances using different compile-time customizations must be used. When using multiple instance types, there are two possible approaches for managing customizability:

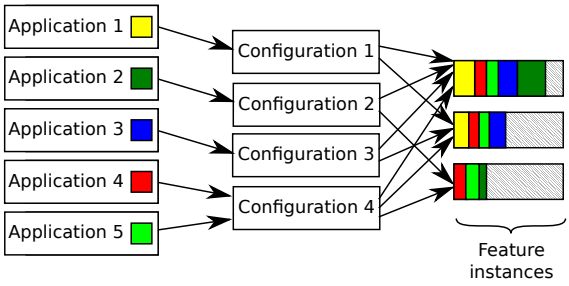
- *Application-Based Binary (ABB) approach:* Using the ABB approach, custom application binaries are generated for every application variant that is used by tenants. Resources can be shared between application instances when the users have identical customizations. Instances of every used application variant must always be active in the cloud environment, even if there are no users, to ensure new users requests can be handled with acceptable quality of service[1]. Provided the number of different customizations is limited, this approach can be acceptable, but as the number of application variants increases the cost of using this approach increases as well.

- *Feature-Based Binary (FBB) approach:* In the feature-based approach, an application is split into multiple interacting components. These components are implemented as services that provide a distinct part of the application functionality. Service instances are associated with individual features, and the application is composed out of these resulting feature instances by using a SOA. Individual feature instances provide a specific, well-defined part of the application functionality, and as all applications making use of a feature make use of the same customizations, resources can be shared within these instances. Using this approach the customization is achieved by changing the services that are active and by the way in which the services are composed.

[1]If no instance would be active, a new instance would have to be created when a user request is received. As creating new application instances may require considerable time, this would result in unacceptable performance.



(a) Load on application instances created using the ABB multi-tenancy approach.



(b) Load on application instances created using the FBB multi-tenancy approach.

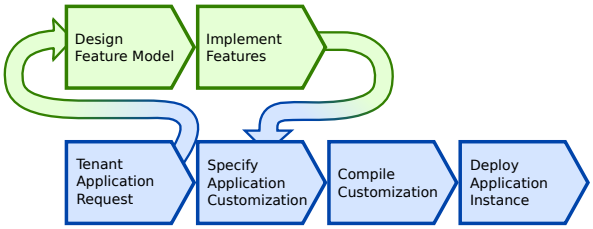
Figure 1: An illustrative example comparing the ABB and a FBB multi-tenancy approaches.

Both approaches still support some multi-tenancy: in the ABB approach this is by sharing resources between identically customized application, while in the FBB approach this is done by ensuring the feature instances themselves are multi-tenant. Figure 1 shows an illustrative example of both approaches. In this example, there are five applications, of which two make use of an identical feature configuration. In the ABB approach this results in four different instance types, one for every configuration. When the FBB approach is used, the number of instances is not dependent on the number of configurations, but rather on the number of different feature instances; in the example, we assume the application is composed out of three different feature instances. In the sample scenario, there are more application variants than there are features, which is why the FBB approach results in more multi-tenancy.

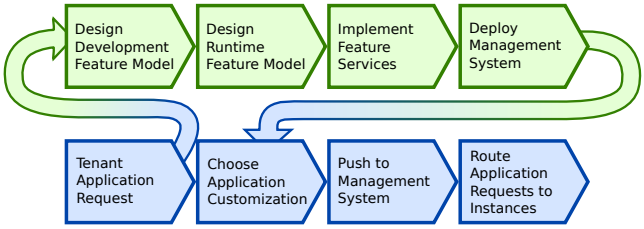
In practice, it is often the case that there are more potential application variants than there are application features, in this case making it preferable to a feature-based approach. The FBB approach can also be modified to incorporate some properties of the ABB approach. This results in a hybrid approach that can support both feature-based and application-based instances. We will discuss this hybrid approach later in this section.

3.1 Application development, deployment and management

The processes for developing, deploying and managing ap-



(a) The ABB development and deployment process.



(b) The FBB development and deployment process.

Figure 2: The processes for developing and deploying multi-tenant feature-based SaaS applications using the ABB and FBB approaches.

plications differ between both approaches. Figures 2a and 2b show the processes for respectively the ABB and FBB approaches. When developing and deploying new applications, the topmost processes are executed. When new applications are instantiated for a tenant, the second processes are used. In some cases, new features may still have to be implemented for specific very large tenant organizations when they request new application instances, making it possible for the deployment workflow to be interrupted by an additional development phase.

3.2 Feature models

Feature models are be used at various times during application development and deployment. In this paper, we focus on four relation types that are structured in a hierarchy:

- **Mandatory(a, b)**: If a feature **a** is included, the feature **b** must be included as well.
- **Optional(a, b)**: If a feature **a** is included, the feature **b** may be included. Conversely, the feature **b** must not be included if **a** is not included.
- **Alternative(a, S)**: If a feature **a** is included exactly one of the features contained in the set **S** must be included. If **a** is not included, none of the features in **S** may be included.
- **Or(a, S)**: If a feature **a** is included, at least one of the features contained in the set **S** must be included. If **a** is not included, none of the features in **S** may be included.

3.3 ABB applications

The process for developing ABB applications is straightforward, as shown in Figure 2a: first a feature model is defined, then the features are implemented. For this process traditional Software Product Line Engineering (SPLE) approaches can be used.

Deploying applications is done in multiple steps:

1. The process starts when a tenant requests a new application or requests changes to an existing application. For new tenants, this request is typically made by a seller employed by the platform provider. It may also be possible for existing clients to make these requests themselves using a configuration interface. When new organizations are added, it is possible for them to have new requirements that are not yet supported by the current application. If this is the case, and depending on the potential profit, a management decision may be made to implement these requirements, in which case the development process is started to first update the application by adding the additional features.

2. Next, a tenant configuration interface is used to specify the application configuration. This interface can be used to specify the features that are included in the tenant application and their configuration.

3. Once the customizations are selected, it is possible to compile the application instance. This is only needed if no instance with an identical configuration exist.

4. The application instance is deployed in the cloud environment. Managing applications can then be done using standard management techniques for multi-tenant applications such as [14]. If an identical application[2] already exists, the application instance is not allocated; instead, the management system reconfigures the existing instance to offer it to the new tenant using multi-tenancy.

3.4 FBB application development

Developing applications using the FBB is a process that requires four steps as shown in Figure 2b:

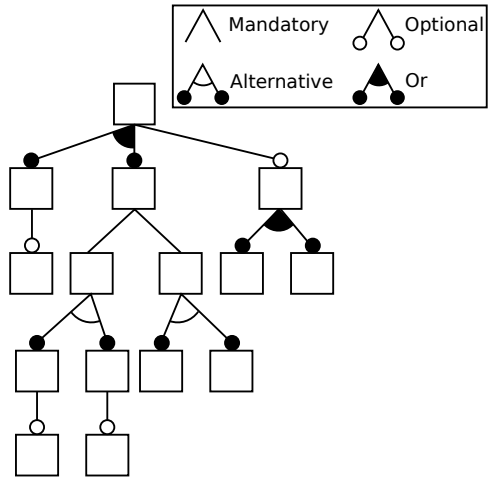
1. First, an application feature model is defined. This model contains all the features that are defined by the application, and thus encompasses all the customization and configurations that a tenant may request.

2. The development model can then be analyzed to determine how each of the features can be implemented. This can be done by classifying them into compile time and runtime features. The former features are features that impact performance or require distinct code modules, and therefore require distinct application instances at runtime, while the latter features can be implemented by configuration or AOP changes to an instance at runtime. Based on this, a model where all runtime features are removed can be determined. All of the features in this model are thus associated with specific code modules.

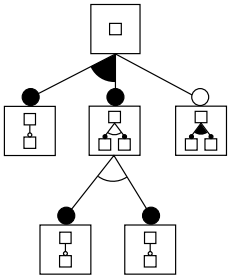
3. The services that are defined in the runtime feature model can then be implemented.

4. Finally, the service binaries and runtime feature model are pushed to the management system running within the cloud environment. Once this is done, it is possible

[2]In this case an identical instance is one which contains the same static customizations. Two identical instances may still differ in functionality if they have different runtime weaving or configuration changes.



(a) A development feature model containing compile time and runtime variability.



(b) A collection of services that each deliver part of the functionality of an application. Features may or may not be customizable themselves (illustrated here by showing smaller feature models within the features), but this customization must be provided using runtime changes only.

Figure 3: An illustrative example of a development and runtime feature model. Both models offer the same customizations, but in the development model all features are contained within the model while the runtime model contains only compile-time customizations requiring separate feature instances. The runtime changes are handled by ensuring the feature instances themselves are customizable as well.

to deploy the application services, but there no are applications making use of the newly developed features yet.

An important and complex step in the application development is the removal of runtime changes from the development feature model. To achieve this, all of the features that can be provided at runtime are stripped from the model, resulting in a smaller runtime feature model containing only the features for which separate instances are needed. Each of these features can then be implemented as a separate instance that realizes specific application functionality. These feature instances may themselves be customizable, but these customizations may not require compile-time changes. The model transformation is illustrated in Figure 3, where we show how an example development feature model (Figure 3a) may be transformed into a runtime feature model (Figure 3b) where some features are realized by providing them using different service instances while others are realized by runtime variation of instances at runtime.

This model transformation can be a manual step during application development, but for changes that can be represented using AOP aspects that need to be applied to components, this approach can be automated. We described an approach for automated feature conversion in [8]. In this approach it is possible to automate the conversion of a feature model containing code modules and aspects applying to specific components to a runtime feature model where all application features refer to a code module.

In the presented approach, every feature in the runtime feature model is associated with a code module that is used to instantiate the feature. Some features may however be defined purely to add structure to the feature model. To make this possible, developers may define *empty* features. These features are not associated with code modules and including them does not create new instances.

3.5 FBB application deployment

Once the application is developed, new instances can be deployed for clients. Typically, this process is done in the four steps shown in the second process in Figure 2b:

1. First, a tenant may requests a new application or modifications to an existing application. Like for ABB applications this request may be processed at once if all of the requested features already exist. Alternatively, this may also lead to a new development cycle where additional features are defined and implemented.

2. A tenant configuration interface is used to specify the application configuration. This interface is based on the development feature model (containing all of the changes) and may be generated automatically based on the feature model.

3. The new application configuration is then pushed to the management system. This management system then allocates resources on existing feature instances or instantiates additional feature instances to accommodate the application.

4. The application is then deployed and available.

An advantage of using a runtime feature model to represent application components is that it is possible to define open variation points [6]: features not just be either selected or excluded, but they may also remain undecided. This makes it possible to defer some decisions until runtime, reducing resource costs.

3.6 FBB application management

The cloud management system is responsible for allocating resources for the various feature instances that must be deployed in the FBB approach. The runtime feature model is known by the cloud management system, and is used by the management system to determine the features that are included in application. The final feature configuration of an application is dependent on the feature model, selected features, excluded features, and how open variation points are filled in by the management system.

Due to the presence of open variation points, there may be multiple possible feature configurations for an application, which results in interesting opportunities for reconfiguration of applications at runtime. This in turn results in multiple benefits. 1) It is possible to reduce the number of

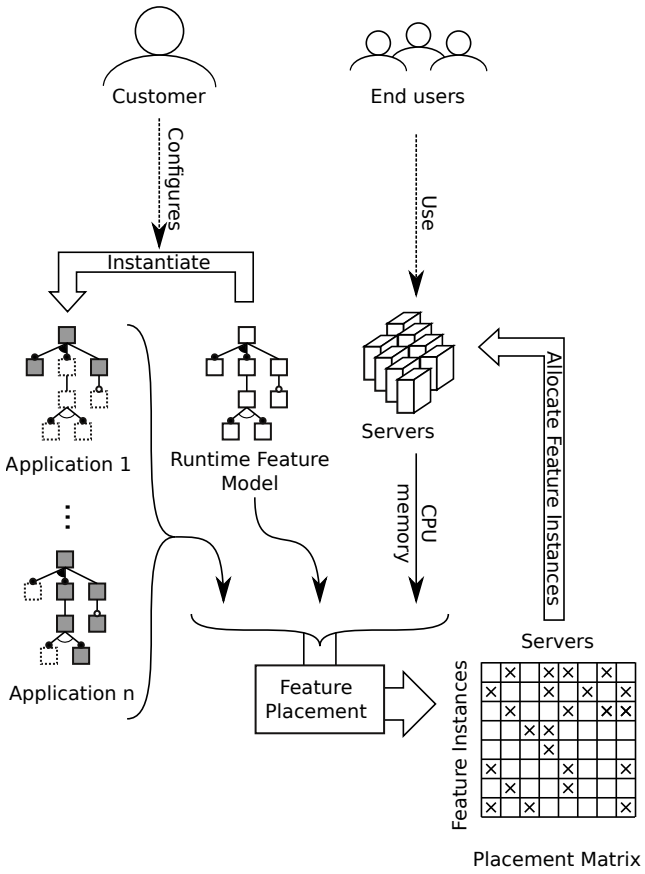


Figure 4: The FBB approach management system. The feature placement algorithm is responsible for allocating resources for the various application instances.

active instance types by preferring configurations that are already deployed within the cloud system. This increases multi-tenancy and reduces resource costs. 2) Similarly, it is possible to choose a feature configuration that results in the quickest deployment, which can also be done by reusing feature instances that already exist and minimizing the number of new features that must be deployed.

To achieve these benefits, it is important to make use of management algorithms that are feature-aware. These management algorithms combine cloud application placement algorithms [1, 14, 15], which are used to allocate resources in clouds, with feature-awareness. The resulting feature placement algorithms can both determine an optimal application feature configuration and the placement of feature instances on servers in the cloud environment.

Figure 4 shows the how the feature placement algorithm functions. The feature placement is aware of the servers that are active within the cloud datacenter, the feature model, and the applications that make use of this feature model. Based on this information, a feature configuration can be determined for the applications, and an allocation can be determined indicating which feature instance is allocated on which server for which application. This application placement is then executed on the servers. We have previously designed multiple feature placement algorithms and discussed them in-depth in [9, 10].

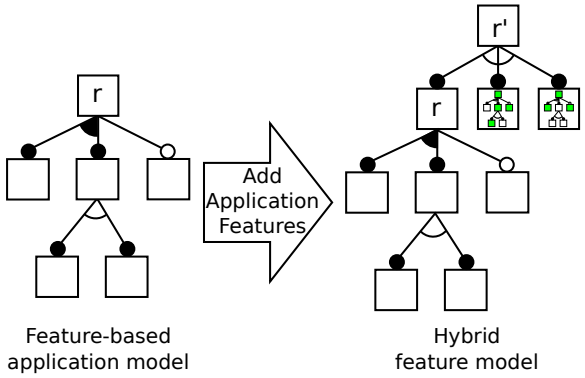


Figure 5: By adding specific application customizations as separate instances to the feature model, a hybrid approach using both FBB and ABB approaches can be used.

3.7 Hybrid multi-tenancy

As mentioned previously, the FBB approach results in fewer application instance types provided that the number of features is lower than the possible number of different application variations. In some cases it may however be preferable to use ABB applications rather than FBB applications:

1. If the number of custom variants is lower than the number of features, using the FBB approach would result in more different instance types compared to the application-based approach.

2. If a specific variant has a very large number of users, or has many tenants using it. In this scenario it may be preferable to create a single instance implementing the variant as this results in lower network demand and communications, and thus conversely may result in lower costs and higher performance.

3. If a specific variant may only be used by multiple end users of the same tenant due to security reasons. In this scenario deploying separate instances for every feature results in more active application than in a scenario where a single instance is generated.

It is possible to extend the feature-based approach to also support ABB application variants. This can be achieved by adding a root node r' as a parent of the original root node r of the feature model. The set of specific application instances I is then made. Every instance $i \in I$ is a feature that links to a code module containing the entire application with specific set of customizations as a single instance. By adding the original root feature of the feature model r' and the instance features I as using an alternative relation, a new feature model can be constructed. This feature model contains all the relations of the original feature model with the addition of the new **Alternative**$(r', I \cup \{r\})$ relation. Using this approach, a model is constructed where every instance is either an instance of the feature-based approach, of of one of the chosen ABB instances.

This hybrid approach, further illustrated in Figure 5 combines properties of both the FBB and ABB approaches. The hybrid multi-tenancy approach still results in a runtime feature model with multiple services, and is thus equivalent to

the FBB approach with a larger feature model. By combining hybrid application models with feature placement algorithms and open variation points interesting synergies can be discerned: the feature placement algorithm can decide the optimal application configuration by either deploying the application as a single instance or by deploying the application as a collection of feature instances depending on which option results in the lowest resource cost.

4. ANALYSIS

We compare the ABB, FBB and hybrid approaches theoretically by comparing the number of instance types that may be generated by them. As discussed in the previous section, resources can only be shared between identical instances. Therefore, there will be less multi-tenancy when more different instance types may be generated, making it preferable to have fewer possible instance types. We refer to the maximum number of instances that may be generated by a management approach for a feature model as the Maximum number of Instance Types (MIT). Note that the MIT results in a worst case scenario for a given feature model, and the number of instances active at any time may be lower: in the ABB approach there will never be more different instance types than there are applications and in the FBB the management algorithm may exploit open variation points to reduce the number of instantiated instance types.

4.1 ABB instance count

In the ABB approach, the MIT is limited by the possible number of application variants as every variant is statically compiled and then allocated as a separate instance. Thus, to determine the potential number of different instances, the total number of valid feature selections of the application feature model must be determined. We represent the MIT in the ABB approach of a family of applications with feature model F by $\mathcal{A}(F)$.

$\mathcal{R}$ represents the collection of all relations within the feature model, and $\mathcal{R}(f,\cdot)$ represents all relations in $\mathcal{R}$ with parent feature f. In our analysis we focus, as stated previously, on feature models consisting of four types of relations: **Mandatory**, **Optional**, **Alternative** and **Or**. For a feature x, $\mathcal{A}^f(x)$ represents the MIT of the feature model containing x as the root feature and all of the subfeatures of w within the original feature model. For a relation x, $\mathcal{A}^r(x)$ represents the number of variations introduced by this specific relation. The total MIT is then computed by determining the number of variants of the root feature r, implying that $\mathcal{A}(F) = \mathcal{A}^f(r)$.

When a feature f only has a single relation r in which it is the parent, $\mathcal{A}^f(f) = \mathcal{A}^f(r)$. It may however occur a feature is a parent in multiple relations. In this case, each of these relations results in a collection of possible variants. As all of these relations are independent, the total number of variants can be computed combinatorially by multiplying the individual MIT counts. This is expressed formally in Equation (1).

$$\mathcal{A}^f(a) = \prod_{r \in \mathcal{R}(a,\cdot)} \mathcal{A}^r(r) \tag{1}$$

The MIT counts can be computed for the various relation types as follows. **Mandatory** relations do not cause additional variants as they must always be included if the parent feature is included. The child feature c will however itself result in multiple variations $\mathcal{A}^f(c)$, which is expressed in Equation (2). **Optional** relations either result in including the child feature (resulting in all possible variants of the child feature c, $\mathcal{A}^f(c)$) or in not including the child feature (resulting in an additional configuration and increasing the number of variants by 1); this is expressed in Equation (4). **Alternative** relations always result in exactly one child feature being included resulting in causing all of the variability of its child nodes to be included, which is expressed in Equation (4). The formula for the **Or** relation, shown in Equation (5), can be easily derived from Equations (1) and (3) by observing that an **Or** is equivalent to a collection of **Optional** relations where one case, that where none of the child features are included, is removed.

$$\mathcal{A}^r(\mathbf{Mandatory}(\mathbf{a},\mathbf{b})) = \mathcal{A}^f(b) \tag{2}$$

$$\mathcal{A}^r(\mathbf{Optional}(\mathbf{a},\mathbf{b})) = \mathcal{A}^f(b) + 1 \tag{3}$$

$$\mathcal{A}^r(\mathbf{Alternative}(\mathbf{a},\mathbf{S})) = \sum_{s \in S} \mathcal{A}^f(s) \tag{4}$$

$$\mathcal{A}^r(\mathbf{Or}(\mathbf{a},\mathbf{S})) = \left(\prod_{s \in S} \left(\mathcal{A}^f(s) + 1 \right) \right) - 1 \tag{5}$$

4.2 FBB instance count

For feature based applications the MIT, which is represented by $\mathcal{F}(F)$, is limited by the number of features. Therefore, $\mathcal{F}(F)$ equals the number of features in the feature model F. This value can also be be computed making use of the feature hierarchy, similar to how this was done for application-based multi-tenancy, which is useful for comparing the theoretical performance of FBB with ABB. $\mathcal{F}(F)$ can be easily computed based on the same approach we previously used to compute MIT for the ABB approach. The MIT of a feature x is represented as $\mathcal{F}^f(x)$; the MIT of a relation x is represented by $\mathcal{F}^r(x)$. $\mathcal{F}(F) = \mathcal{F}^f(r)$ with r the root feature. The formulations for the various relation types can be computed trivially based on their definition:

$$\mathcal{F}^f(a) = 1 + \sum_{r \in \mathcal{R}(a,\cdot)} \mathcal{F}^r(r) \tag{6}$$

$$\mathcal{F}^r(\mathbf{Mandatory}(\mathbf{a},\mathbf{b})) = \mathcal{F}^f(b) \tag{7}$$

$$\mathcal{F}^r(\mathbf{Optional}(\mathbf{a},\mathbf{b})) = \mathcal{F}^f(b) \tag{8}$$

$$\mathcal{F}^r(\mathbf{Alternative}(\mathbf{a},\mathbf{S})) = \sum_{s \in S} \mathcal{F}^f(s) \tag{9}$$

$$\mathcal{F}^r(\mathbf{Or}(\mathbf{a},\mathbf{S})) = \sum_{s \in S} \mathcal{F}^f(s) \tag{10}$$

4.3 Hybrid instance count

The hybrid multi-tenancy approach extends the FBB approach and adds an additional root element r', an **Alternative** relation, and at least one feature linked to an instance of a specific application customization. Therefore the hybrid multi-tenancy approach will always result in at least two more instance types than the FBB approach. Thus, the number of instance types $\mathcal{H}(F)$ of the hybrid approach can be computed by $\mathcal{H}(F) = \mathcal{F}(F)+n+1$ where n is the number of added ABB applications. In the evaluation and analysis, we will focus mainly on the ABB and FBB approaches as the hybrid approach can be considered as a special case of the

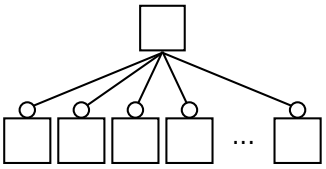


Figure 6: The worst case feature model for the ABB approach resulting in the maximum possible instance type count.

FBB approach with a additional features and thus a slightly higher MIT.

4.4 Worst-case comparison

FBB and ABB multi-tenancy behave very differently depending on the feature models they are used with. The FBB approach always results in the same number of features for every feature model irrespective of the relations within the model as it is only dependent on the number of features within the model. Therefore, we consider the worst case for this approach to be the one where the model only results in few variations while requiring many features. For a feature model with n features containing only **Mandatory** features, FBB results in n different feature instance types, while ABB results in an MIT of only 1 as there is no customization. In this case, it would be preferable to restructure the feature model to reduce the number of features, possibly reducing the feature model to a single feature. This is however an edge case as there is no customizability in this scenario. Ignoring **Mandatory** relations, the **Alternative** relation performs worst for the FBB approach compared to the ABB approach. A feature model containing only a single **Mandatory(r, S)** relation results in $|S|+1$ instance types in the FBB approach and only in $|S|$ variants in the ABB approach.

The worst case scenario for the ABB approach is when it is used for a flat feature model containing only **Optional** relations, as illustrated in Figure 6. This model results in 2^{n-1} possible variants with n the number of features within the model.

THEOREM 1. *The worst case feature model, resulting in the maximum MIT for a given number of features, only contains **Optional** relations.*

PROOF. We assume that a model F exists with the maximum $\mathcal{A}^f(r)$ that contains relations other than **Optional** relations. This implies there must be at least one relation r that is not an **Optional** relation. This relation can be one of three types:

1. **Mandatory(a, b)**: By replacing **Mandatory(a, b)** with **Optional(a, b)**, $\mathcal{A}^r(r)$ can be increased. This in turn increases the model MIT.

2. **Or(a, S)**: In this case, $\mathcal{A}^r(r) = \left(\prod_{s\in S}\left(\mathcal{A}^f(s)+1\right)\right) - 1$. By replacing the **Or** relation by a set of **Optional(a, s)** relations for every $s \in S$, the number of variants will be increased by one, increasing $\mathcal{A}^f(a)$ and in turn increasing the MIT of the new feature model.

3. **Alternative(a, S)**: From Equations (4) and (5), and the fact that the set S must always contain at least two features, it can be concluded that replacing the relation **Alternative(a, S)** by **Or(a, S)** the MIT can be increased.

In each of the cases a new feature model F' can be constructed for which $\mathcal{A}(F) < \mathcal{A}(F')$, contradicting the assumption. □

THEOREM 2. *The feature model resulting in the maximum MIT for a given number of features is flat and consists of a root feature r and a set of relations **Optional(r, s)** for all features $s \in \{F/r\}$.*

PROOF. Suppose the contrary that a feature model F exists that is not flat and that results in a higher MIT. This model must only contain **Optional** relations as proven in Theorem 1. This model must therefore contain a relation **Optional**(n^r, a), with n^r the root node, and a feature a that is itself parent in one or more **Optional**(a, s) relations with $s \in S$ as child features. The contribution to the total MIT by the feature a and its subfeatures, represented as C^a can be determined using Equations (1) and (3):

$$C^a = \mathcal{A}^r(\mathbf{Optional}(\mathbf{a},\mathbf{s})) = 1 + \prod_{s\in S}\left(A^f(s)+1\right)$$

An alternative feature model F' can be constructed where the features $s \in S$ are attached directly to the root node instead of to the feature a. The contribution of a and the features in S is then represented by C'^a (in this model, a no longer has child nodes, ensuring $\mathcal{A}^f(a) = 1$):

$$\begin{aligned} C'^a &= (\mathcal{A}^f(a)+1) \times \prod_{s\in S}\left(A^f(s)+1\right) \\ &= 2 \times \prod_{s\in S}\left(A^f(s)+1\right) \end{aligned}$$

As $A^f(f) \geq 1$ for all features f, we can conclude that $C^a < C^b$, which due to Equation (1) ensures that $\mathcal{A}(F) < \mathcal{A}(F')$. Therefore, this new model results in more variations than the original model F, contradicting the assumption that the feature model F resulted in the maximum MIT given the number of features. □

From this analysis, we conclude that if a model has many **Optional** or **Or** relations, or if a single feature is the parent in many relations, the FBB approach will generally result in fewer instance types than the ABB approach. **Mandatory** and **Alternative** relations may however work better in an ABB approach.

5. EVALUATION RESULTS

As noted in the previous section, it is trivial to construct feature models where one approach is better than the other. It is however important to determine when which approach is preferable for realistic feature models. For our evaluations we make use of three feature models of commercial SaaS applications, a composed feature model, and randomly generated feature models based on the structure of the previous models.

We use the feature models of thee commercial SaaS applications as a baseline for our evaluations. The models are that of a Medical Communications (MC) application, a Document Processing (DP) applications and a Medical Data Management (MDM) application. These models contain 12, 22 and 16 features respectively. Based on the application feature models we also define a composed feature

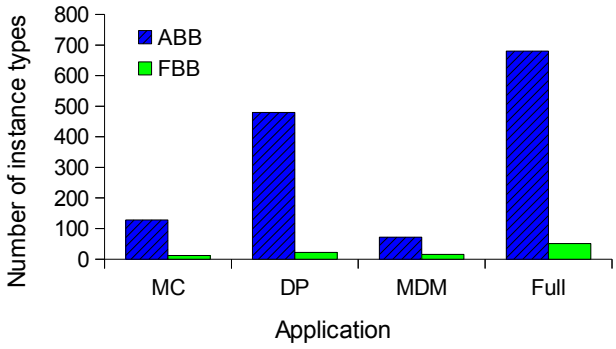


Figure 7: The MIT for the ABB and FBB approaches for three commercial applications, *MC*, *DP* and *MDM*, and a composed model containing all three models, *Full*.

model *Full* which contains all of the features of the *MC*, *DP* and *MDM* models composing them using an **Alternative** relation. This represents the situation where all of the previous applications are executed on the same application platform. The composed model contains in total 51 features.

We also defined a collection of randomly generated feature models of varying sizes to compare the approaches for feature models of differing sizes. The models are generated ensuring they are similar to the *MC*, *DP*, *MDM* and *Full* models in terms of structure and frequency of relation types. First the set of features is generated and one feature is selected as root of the feature model tree. Next the other features are iteratively added to the feature model by selecting a random node as the parent and one or more of the remaining features are added as child nodes. We use an equal probability for selecting any of the relation types, with **Optional** and **Mandatory** relations having (by definition) one child while **Alternative** and **Or** relations have between 2 and 6 features as child nodes (chosen uniformly at random).

Figure 7 compares the number of instance types that may have to be deployed within a cloud environment for the four application feature models. For the four cases the MIT is lower for the FBB approach than it is for the ABB approach. The largest difference is observed for the *DP* where the ABB results in 22 times the number of possible instances while the *MDM* case results in the smallest increase (4.5 times as many possible instances as the FBB approach).

By using randomly generated feature models, we can further evaluate the behavior of both approaches. Figure 8 compares the number of variants for the ABB and FBB approaches for varying numbers of features. For the FBB approach the number of instance types always equals the number of features, for the ABB approach the number of instance types depends on the used feature model. For every data point 10000 randomly generated feature models were used, which results large spread of resulting values. Figure 9 shows the distribution of the number of instance types for the evaluation.

As the number of features increases, the FBB approach results in much fewer instance types than the ABB approach. On average, the ABB approach only results in fewer instance types for feature models containing 4 or less features, but the average is heavily skewed by outliers. For the median feature

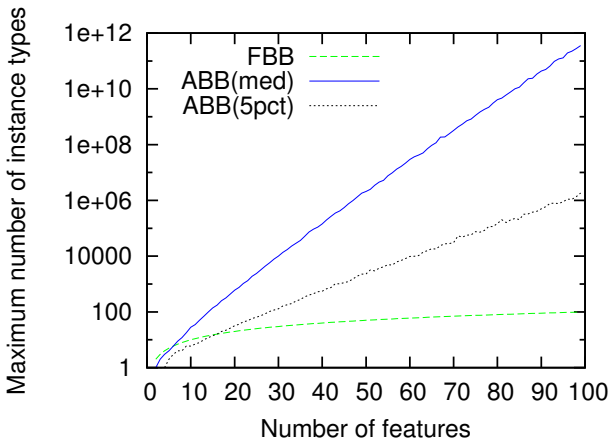


Figure 8: The median of MIT for the ABB and FBB approaches for varying numbers of features (10000 feature models per data point). The 5th percentile of the ABB values is shown to give an indication of the distribution of the number of instance types for the ABB approach.

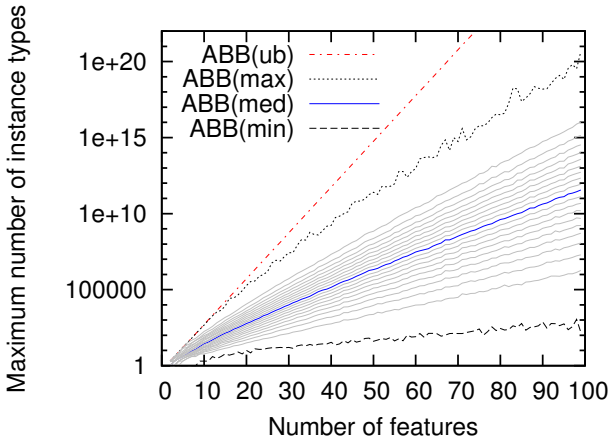


Figure 9: The distribution of MIT for the ABB approach for varying numbers of features (10000 feature models per data point). The theoretical upper bound (ub), maximum (max), median value (med) and minimum (min) are shown. The gray lines show the percentiles of the distribution (5th until 95th in increments of 5).

model, the ABB approach performs best for models with less than 7 features. For larger models, the FBB approach results in fewer variants.

6. DISCUSSION

The analysis and evaluations show that the FBB generally results in fewer distinct application instances compared to the ABB approach, as measured by the MIT metric. For very small feature models containing 6 or fewer features, the ABB approach may however in be preferable. If the feature model contains many mandatory and alternative relations, the ABB approach may also result in a lower MIT value.

The hybrid approach is an extension to the FBB approach,

adding specific application configurations implemented using the ABB approach. Because of this, the hybrid approach can be used in all scenarios where the FBB approach may be used, resulting in a slightly higher MIT (depending on the number of application configurations that are added using the ABB approach). In a scenario where the ABB results in fewer instances than FBB, however, these variants can be added to the model, resulting in fewer instantiated instances at runtime compared to the FBB approach. The hybrid approach may also be useful for application configurations that are used by many tenants if the single-instance application requires fewer resources, which may be the result of a reduction in communication overhead.

The most important disadvantage of the FBB approach is that it becomes more difficult to manage applications, as more communication between components must be taken into account. This disadvantage is shared with the hybrid approach. A second disadvantage of the FBB approach is that more application instances are needed to offer an application, which may be disadvantageous if some components are rarely used causing the instances to be underutilized. This problem can be mitigated using the hybrid approach as, in such cases dedicated instances can be defined.

7. CONCLUSIONS

In this paper, we formalized two approaches for managing variability in multi-tenant SaaS environments: FBB, a SOA-based which composes the application out of multiple multi-tenant components and ABB which statically generates multiple distinct multi-tenant applications based on a common feature model. We described how applications can be developed, deployed and managed using both approaches. We also presented a hybrid approach, combining beneficial properties of both the ABB and FBB approaches.

We found that, for feature models with more than 6 features requiring compile-time changes, the FBB approach results in fewer possible runtime instance types, which in turn results in more opportunities for exploiting multi-tenancy and lower costs. For models with fewer features, the ABB approach will perform better.

8. ACKNOWLEDGMENTS

Hendrik Moens is funded by the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT).

9. REFERENCES

[1] C. Adam and R. Stadler. Service Middleware for Self-Managing Large-Scale Systems. *IEEE Transactions on Network and Service Management*, 4(3):50–64, Dec. 2007.

[2] L. Baresi, S. Guinea, and P. Liliana. Service-Oriented Dynamic Software Product Lines. *Computer*, 45(10):42–48, 2012.

[3] R. Capilla, J. Bosch, P. Trinidad, A. Ruiz-Cortés, and M. Hinchey. An overview of Dynamic Software Product Line architectures and techniques: Observations from research and industry. *Journal of Systems and Software*, 91:3–23, May 2014.

[4] S. Hallsteinsen, M. Hinchey, and K. Schmid. Dynamic Software Product Lines. *Computer*, 41(4):93–95, Apr. 2008.

[5] M. Hinchey, S. Park, and K. Schmid. Building Dynamic Software Product Lines. *Computer*, 45(10):22–26, 2012.

[6] R. Mietzner, A. Metzger, F. Leymann, and K. Pohl. Variability modeling tosupport customization and deployment of multi-tenant-aware Software as a Service applications. In *ICSE Workshop on Principles of Engineering Service Oriented Systems*, volume 215483, pages 18–25. IEEE, May 2009.

[7] R. Mietzner, T. Unger, R. Titze, and F. Leymann. Combining Different Multi-tenancy Patterns in Service-Oriented Applications. In *2009 IEEE International Enterprise Distributed Object Computing Conference*, pages 131–140. IEEE, Sept. 2009.

[8] H. Moens, E. Truyen, S. Walraven, W. Joosen, B. Dhoedt, and F. De Turck. Developing and Managing Customizable Software as a Service Using Feature Model Conversion. In *Proceedings of the 3rd IEEE/IFIP Workshop on Cloud Management (CloudMan)*, pages 1295–1302, 2012.

[9] H. Moens, E. Truyen, S. Walraven, W. Joosen, B. Dhoedt, and F. De Turck. Feature Placement Algorithms for High-Variability Applications in Cloud Environments. In *Proceedings of the 13th Network Operations and Management Symposium (NOMS 2012)*, pages 17–24, 2012.

[10] H. Moens, E. Truyen, S. Walraven, W. Joosen, B. Dhoedt, and F. De Turck. Cost-Effective Feature Placement of Customizable Multi-Tenant Applications in the Cloud. *Journal of Network and Systems Managemement*, Feb. 2013 (in press).

[11] B. Morin, F. Fleurey, N. Bencomo, J.-M. Jézéquel, A. Solberg, V. Dehlen, and G. Blair. An Aspect-Oriented and Model-Driven Approach for Managing Dynamic Variability. In *Model Driven Engineering Languages and Systems*, pages 782–796. Springer Berlin Heidelberg, 2008.

[12] C. Parra, X. Blanc, and L. Duchien. Context awareness for dynamic service-oriented product lines. In *Proceedings of the 13th International Software Product Line Conference (SPLC 09)*, pages 131–140. Carnegie Mellon University, Aug. 2009.

[13] W. Sun, X. Zhang, C. J. Guo, P. Sun, and H. Su. Software as a Service: Configuration and Customization Perspectives. In *IEEE Congress on Services Part II (services-2)*, pages 18–24. IEEE, Sept. 2008.

[14] C. Tang, M. Steinder, M. Spreitzer, and G. Pacifici. A scalable application placement controller for enterprise data centers. In *Proceedings of the 16th international conference on World Wide Web*, pages 331–340, 2007.

[15] B. Urgaonkar, A. L. Rosenberg, and P. Shenoy. Application Placement on a Cluster of Servers. *International Journal of Foundations of Computer Science*, 18(05):1023–1041, 2007.

[16] S. Walraven, D. Van Landuyt, E. Truyen, K. Handekyn, and W. Joosen. Efficient customization of multi-tenant Software-as-a-Service applications with service lines. *Journal of Systems and Software*, 91:48–62, May 2014.