Developing and Managing Customizable Software as a Service Using Feature Model Conversion

Hendrik Moens*, Eddy Truyen†, Stefan Walraven†, Wouter Joosen†, Bart Dhoedt* and Filip De Turck*
* Ghent University, IBBT Department of Information Technology
Gaston Crommenlaan 8/201, B-9050 Gent, Belgium
† Katholieke Universiteit Leuven, DistriNet Research Group, Dept. Computer Science
Celestijnenlaan 200A, B-3001 Heverlee, Belgium
e-mail: hendrik.moens@intec.ugent.be

Abstract—In recent years, there has been a growing interest in cloud technologies. Using current cloud solutions, it is however difficult to create customizable multi-tenant applications, especially if the application must support varying Quality of Service (QoS) guarantees. Software Product Line Engineering (SPLE) and feature modeling techniques are commonly used to address these issues in non-cloud applications, but these techniques cannot be ported directly to a cloud context, as the common approaches are geared towards customization of on-premise deployed applications, and do not support multi-tenancy.

In this paper, we propose an architecture for the development and management of customizable Software as a Service (SaaS) applications, built using SPLE techniques. In our approach, each application is a composition of services, where individual services correspond to specific application functionalities, referred to as features. A feature-based methodology is described to abstract and convert the application information required at different stages of the application life-cycle: development, customization and deployment. We specifically focus on how development feature models can be adapted ensuring a one-to-one correspondence between features and services exists, ensuring the composition of services yields an application containing the corresponding features. These runtime features can then be managed using feature placement techniques. The proposed approach enables developers to define significantly less features, while limiting the amount of automatically generated features in the application runtime stage. Conversion times between models are shown to be in the order of milliseconds, while execution times of management algorithms are shown to improve by 5 to 17% depending on the application case.

Index Terms—Clouds, SaaS, SPLE, Design methodology

I. INTRODUCTION

While the adoption of cloud technologies is on the rise, limitations prevent their uptake in some markets. Current cloud frameworks do not support the creation of highly customizable Software as a Service (SaaS) applications that retain the multi-tenant nature of these applications, where a single application instance is shared between users. This is especially difficult if both functional and Quality of Service (QoS) variation are required. An approach where changes are statically compiled into the application could be used to run such applications on a cloud, but this would lead to the generation of different applications for clients, loosing the cost-advantages offered by multi-tenancy. Furthermore, the cloud management software would still need extensive support for QoS management to ensure varying non-functional application requirements are met.

To develop customizable applications, Software Product Line Engineering (SPLE) techniques are often used. The software is modeled as a collection of functionalities, which are referred to as features. An application can then be created by combining these features. As it is possible for some features to require other features, or for some features to conflict with others, relations between features must be modeled as well. Feature modeling can become an integral part of the software development process, where code modules, configurations and Aspect-Oriented Programming (AOP) [1] aspects can be created that realize specific features. These feature models can then be used to communicate and configure variability in applications for clients [2]. They can also be used in the deployment process [3], using a Service-Oriented Architecture (SOA) approach where individual features map to services, and in the management and provisioning of these services, using a feature placement algorithm [4] that takes feature relations into account to determine where services implementing these features are placed.

Feature models are suited for many different phases in the life cycle of customizable applications, but maintaining a single feature model that can be used in each of the different phases can be complex, or even impossible. In particular, the representation of information in feature models, as they are specified during application development are not suited as an input for resource allocation algorithms, such as feature placement algorithms. This restriction either forces developers to work using cumbersome restrictions, increases development complexity as a conversion between a development model and a runtime model must be determined, or complicates feature-aware management of multi-tenant applications.

We present an approach where a logical feature model is defined and configured during development using existing SPLE techniques. When the service is deployed, a runtime version of the feature model is generated, together with a mapping between the development feature model and this runtime feature model. Services can then be allocated with
In this paper we describe a system architecture for developing and managing such applications, focusing on the role of feature models within the management system, and on how a mapping between development and runtime feature models can be realized. To evaluate the presented approach, we use the application cases of the CUSTOMSS project. Within the scope of this project a platform for developing and managing highly customizable cloud applications will be designed, with a focus on applications in the domain of document processing, medical information management and medical communication systems. We find that, using the presented conversion, development models can be kept significantly smaller than runtime models, while the overhead introduced by the transformation remains low.

In the next section we will discuss related work. In Section III we will discuss SPLE approaches for the development of SaaS applications. Subsequently, in Section IV we will discuss the management architecture, and the role of feature modeling in it. We will elaborate on the conversion between feature models in a development and an runtime context in Section V, after which we will evaluate the approach in Section VI. Finally, in Section VII, we will discuss our conclusions.

II. RELATED WORK

Within the domain of SPLE, a distinction is commonly made between problem-space and solution-space feature models, with mappings between the two [6], [7]. Our approach is complementary and adds an additional model, a runtime problem model, which is used to manage feature-based services. The runtime model is built using basic transformations applied on the development model, and is validated using a logical representation of feature relations, similar to those used in [8].

Recent work has been done in the field of service workflow variability. [9] focuses on how product lines spanning different organizational domains can be integrated, assuming each of the domains offers varying services. We on the other hand focus on how a single domain can offer these variable services. Closely related, in [10], a policy-based framework for publishing customization options of web services is proposed, enabling clients to build their own customizations. This work however focuses on the specification of variable applications, while we focus on the implementation and runtime aspects of developing customizable SaaS applications.

An approach to build variable SaaS applications is proposed in [11]. To achieve this, the authors however focus on configuration-based changes. We on the other hand focus on customization changes by using a SOA application development approach, making it possible to achieve greater customizability. Design-time variability management of SOA applications is discussed in [12] and [13]. We focus on how these development configurations are managed at runtime. The SOA approach in this paper is similar to that proposed in [14], where a single application consisting of different components is proposed. Our approach, using different service instances is however more flexible, as it ensures services can have different, and sometimes incompatible dependencies.

A framework for native multi-tenancy is presented in [2]. In this work, security, isolation and software variability are considered. The authors focus on customization through configuration, laying the responsibility for managing the complexity of variability with the developer. Our approach on the other hand supports true customization changes using multi-tenant instances. Furthermore, the feature model conversion discussed in this work can be used to simplify the variability management done by the developers.

From a cloud management perspective, the described platform ensures underlying service instances are abstracted, effectively building a Platform as a Service (PaaS) where entire services are managed at a higher level, as discussed in [15]. Many such platforms exist, both commercially and in literature. Our approach can be used to extend existing PaaS platforms, ensuring high-variability applications can be built on top of these platforms.

III. VARIABLE SOFTWARE AS A SERVICE DEVELOPMENT

SPLE techniques are often used to develop highly customizable applications. The variability in applications is modeled as a collection of features, where every feature represents a functionality of an application. These features are then related in a feature model. By including a collection of features in an application, and excluding other features, a specific configuration can be created. If this configuration is valid according to the relations defined in the feature model, it can be used to specify a variant of the software application.

Features can depend on each other, they can be incompatible with each other, or they can relate in different ways. To make the complexity of feature models manageable, they are often specified in a hierarchical way [8], [7], where features have parent features and can only be included in an application when the parent feature is selected. These hierarchical feature models can be represented graphically, for which we use the notation as used in [16].

Four hierarchical relations, defined in Table I are typically used: mandatory, optional, alternative and or. In the table,
we also define two additional relations that can occur between arbitrary features, making it possible to express more complex, non-hierarchical feature relations: the excludes and requires relations.

The different relations can be expressed logically using basic propositional calculus, where in a configuration for a software variant a feature in an application is either included (the value corresponding to the feature is assigned value 1) or excluded (the corresponding value is 0). The logical definition of the different relation types is shown in Table II.

At this point, we need to make a distinction between configuration and customization changes [11]. Configuration changes do not impact the code of applications, and are caused by changes in configuration files or application metadata. An example of a configuration change is the logo that should be displayed for branding an application. Customization changes impact the code of an application, and change the code that must be executed. The inclusion of a feature that adds encryption to a process is an example of a customization change.

To enable configuration of SaaS applications, existing techniques can be used [11]. Customization of SaaS applications can be achieved by splitting an application into separate services, that work together to realize the functionality of the application using a SOA [3], [4]. In this approach, every feature that leads to a customization change is realized by associating it with a multi-tenant service.

Figure 2 illustrates a feature model where all features are linked to a code module and a placement configuration. The placement configuration can be used to add additional restrictions that are to be taken into account when placing the service on infrastructure, for example ensuring that the feature can only be placed within the local data center or on a high-reliability instance. It is possible for two features to have the same code module, but a different placement configuration, resulting in different non-functional run-time behavior, as illustrated for the features e and e'.

Formally we introduce a relation codeMap that contains the relations between a feature and its linked code, and a relation configMap that links a feature to its placement configuration. For the feature model in Figure 2, the expression codeMap(a, a.jar) links the feature a with its code module, and the expression configMap(a, Default) indicates the feature is placed using the default placement configuration.

Within this paper we will make a distinction between development feature models, which contain all variation types, and runtime feature models that retain only customization changes. During runtime, the specified configuration changes are added as application metadata.

TABLE II: Conversion of feature model relations to logical statements.

<table>
<thead>
<tr>
<th>Relation</th>
<th>Conversion</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mandatory(A, B)</td>
<td>A ↔ B</td>
</tr>
<tr>
<td>Optional(A, B)</td>
<td>A → B</td>
</tr>
<tr>
<td>Alternative(A, {B, C})</td>
<td>A ↔ B ∨ C</td>
</tr>
<tr>
<td>Or(A, {B, C})</td>
<td>A → B ∨ C</td>
</tr>
<tr>
<td>Excludes(A, B)</td>
<td>¬(A ∧ B)</td>
</tr>
<tr>
<td>Requires(A, B)</td>
<td>A → B</td>
</tr>
</tbody>
</table>

IV. SYSTEM ARCHITECTURE OVERVIEW

A general overview of the system architecture is shown in Figure 3. The system consists of two major components:
a development platform, and an execution platform. These platforms are connected by means of a service repository. This component contains information of the currently deployed services, the workflows that must be executed, and quality rules concerning these workflows.

In the next sections we will give a general overview of both platforms, followed by an in-depth discussion of the management platform. Finally, we will discuss the role of feature models in the application architecture.

A. Development platform

The development platform contains a service development IDE, which can be used to develop customizable applications. It allows the development of multi-tenant services, and provides the required feature modeling tools to build the application feature model. The IDE also provides a testing environment where the functionality of individual services and composed applications can be evaluated.

A second part of the development platform is the configuration and deployment interface. This component can be used by either tenants or in-house vendors to create composed applications. This component is also responsible for testing the final service composition, as not all combinations can be tested during development. Should any tests fail, feedback concerning the configuration will be reported to the developers.

B. Execution platform

The execution platform is built using three layers. At the base there is a cloud platform layer, which contains an existing Infrastructure as a Service (IaaS) or PaaS platform, and forms the foundation on which the execution platform is built. On top of this, a feature management layer manages the services allocated on the infrastructure. These services implement features selected in the application feature model, and can be used by multiple end users, and in multiple feature compositions. It is the responsibility of the feature management layer to ensure that the right server instances are created on the correct locations, given expected QoS requirements. Finally, a process management layer responds to individual user requests. When a request enters the system, a process, making use of feature instances, is instantiated and executed. The process management layer manages a workflow, composed of the services that are controlled by the feature management layer and running on the cloud platform. Services are chosen by the process management layer, and updated during workflow execution based on monitoring information, ensuring the requested QoS is achieved.

C. Execution platform components

The execution platform, shown in Figure 4, manages and coordinates the communication between multi-tenant feature instances. The feature instances are executed on a cloud platform. By using a combination of these feature instances in a workflow, a complete service can be constructed.

In the feature management layer, a service allocation engine is responsible for placing these feature instances on the cloud platform. The information stored in the service repository is used as an input for a feature placement algorithm [4], which returns a placement indicating which features are executed on which servers, and the amount of resources allocated to each of these features. A monitoring component collects input from the various components in the system, and is used to determine when the service allocation engine should reallocate services. The process management layer consists of four components:

- A load balancer is used to balance incoming requests, and forwards requests to a runtime service manager.
- The runtime service manager stores the location of different service compositions. It also maintains a register of services that are reserved for executing workflows. This information enables the component to predict when QoS requirements risk being violated.
- A service composition generator is used to create and adapt service compositions. This component is used to generate a workflow, and to bind concrete services to separate steps in the workflow. This component is QoS-aware, and can dynamically update the composition based on feedback of the runtime service manager to ensure the QoS demands of the workflow are met.
- An orchestration engine service is a multi-tenant service, running on the cloud platform, that contains a process execution engine. This service uses this execution engine to coordinate the different feature instances, combining them to create the desired process.

D. Role of feature models in the application architecture

Feature models of application are key resources at several points in the architecture:

- During development, the features of the application are modeled, and code and configuration information is associated with the feature model.
- During the specification of applications, a salesman or client fills in the feature model, specifying the features that are included in the application.
- During deployment and execution, the feature model of an application is used to determine an optimal placement
of the service by solving a feature placement problem [4].

While, logically, these different phases make use of the same feature model, the view on this feature model is different:

- The developer sees the entire feature model, and links code modules to specific features.
- Salesmen are only interested in the changes that are externally visible. Implementation issues should be hidden. It is preferable to display the choices using a wizard-like interface as mentioned in [2], or using staged configuration [17] allowing the client to gradually fill in the feature model based on simple choices, without requiring a view on the entire software architecture or feature model.
- At runtime, only customization changes matter, and configuration changes can be completely removed from the feature model. A mapping between features and services must exist. At runtime it is impossible for features impacting the implementation of other features to exist, as only one service can be linked with a given feature. If features impacting others exist, the runtime mode must contain two separate versions of the impacted feature: one where the change is included, and one where it is not.

The properties of the three different feature model types are shown in Table III. Within the subsequent sections, we focus on how the conversion between development feature models and runtime feature models can be implemented.

V. FEATURE MODEL CONVERSION

During development, all application features are included in the feature model. These features can imply both configuration changes and customization changes. At runtime, only customization changes should be considered, as configuration changes do not impact the feature model at a code level. Part of the feature model conversion is thus the removal of configuration changes from the runtime feature model.

A specific property of the feature placement algorithm is that it expects features to be linked to a code module and configuration. When two features share the same code but different configurations, for example a single-tenant and multi-tenant version of a component, this implies that both are considered as different features.

A similar problem is the static addition of AOP aspects: when a feature causes the addition of an aspect to a set of components, the management system must ensure that two versions of the component are created as features, one with, and one without the modifications. This kind of behavior can be required when dynamic weaving of aspects would negatively impact the quality properties of the changed feature instances, decreasing the quality of other work-flows that make use of this modified feature. We call the creation of modified features, based on development-time features but statically including specific changes, either to the code modules or to the placement configuration, feature expansion.

We will define techniques for turning a development feature model, \( D \), into a runtime feature model, \( R \), by applying the two aforementioned transformations. We will also define a mapping between the two feature models, to represent how features selected in the development model correspond to those in the runtime feature model. This is represented using a mapping relation \( \text{Map} : D \rightarrow R \) that maps features from one model to the other.

Initially, before any transformations take place, \( D = \varepsilon \), and \( \text{Map} \) contains for every feature \( f \) the relation \( f \mapsto f \), mapping every feature from the development feature model to its corresponding feature in the runtime feature model.

A. Removal of configuration changes

Removal of configuration changes is easy to achieve by simply turning the chosen feature into a dummy feature with nothing associated with it. This empty feature has no code nor placement information associated with it, so it will not be placed by the feature placement system. The inclusion of this empty feature only impacts the application metadata, ensuring the configuration change is realized.

Using these empty features ensures the logical relations of the feature model with respect to the feature are retained, but it also adds useless features to the feature placement process, increasing its computational cost. Transforming feature models, and thereby removing these useless features, the performance of the placement algorithm can be improved. It is however important that the transformed model is logically equivalent to the original model, implying not all empty features can at all times be removed. We call the process of removing redundant empty features feature elimination.

B. Feature elimination

Specific transformations can be used to, in some cases, eliminate empty features created by the removal of configuration changes discussed above. The elimination of mandatory empty features is shown in Figure 5, and is applicable whenever an
empty feature \( e \) has a mandatory child \( n \) (as shown in the figure), or when a node \( n \) has a mandatory empty child \( e \).

Formally, this transformation replaces any mappings referring to \( e \) in Map with references to \( n \). The mapping \( e \mapsto e \) is for example replaced by \( e \mapsto n \). In the runtime feature model \( R \), all references to \( e \) in the relations are replaced by references to \( n \). Finally, the effects of including \( e \) to the application metadata are added to \( n \). The logical correctness of this transformation follows directly from the logical definition of Mandatory, which in this case ensures that \( e \leftrightarrow n \).

A second transformation, the or elimination, is shown in Figure 6. This transformation can be used to eliminate empty features in Or constructs, if the empty features are themselves parent in an Or relation. The equivalence between both constructs shown in the figure follows directly from the fact that from \( n \leftrightarrow e_1 \lor e_2, e_1 \leftrightarrow a_1 \lor a_2, \) and \( e_1 \leftrightarrow a_1 \lor a_2, \) one can deduce that \( n \leftrightarrow a_1 \lor a_2 \lor b_1 \lor b_2 \). In the runtime model \( R \), all Excludes and Requires relations referring to any of the removed nodes, are removed, and replaced by an equivalent relation for each of the children.

Applying the or elimination, unlike mandatory elimination, does remove information, as it becomes impossible to select the features \( e_1 \) in a development model, as no corresponding feature in the runtime model remains. This limitation is taken care of by introducing a set of artificial features. This set represents features that are logically needed for the structuring of the model, but that can not be selected. These features are also removed from Map.

The or elimination as presented here can be generalized for arbitrary amounts of children. An identical transformation can be used, where all Or relations are replaced by Alternative relations.

C. Feature expansion

It is possible that, in the development feature model \( D \), the inclusion of a feature \( a \) changes the implementation of other features, ensuring either the code module linked to the feature changes, or causing the placement configuration of the feature to change. This approach can be used to add, for example a security feature: this can cause changes to the service implementation and can add server placement restrictions.

We introduce a construct FeatureChanges(\( a, f \)), which can be used in the \( D \) model, indicating that the addition of a feature \( a \) changes the implementation of a feature \( f \). Within the \( R \) model, this relation can not exist, and must be removed.

The FeatureChanges relation is converted in the runtime feature model by creating two features \( f' \) and \( f_a \). The mapping codeMap(\( f, c \)) in \( R \) is replaced by a mapping codeMap(\( f', c \)) and an additional mapping codeMap(\( f_a, c_a \)) is added. In this case, \( c_a \) represents the code where the change caused by the feature \( f \) has been included. The configMap relation is changed similarly.

When the FeatureChanges(\( a, f \)) relation is included, this implies that three relations must hold in the runtime feature model:

\[
\begin{align*}
\text{Equation (1)}: & \quad a \land f \leftrightarrow f_a \\
\text{Equation (2)}: & \quad \neg a \land f \leftrightarrow f' \\
\text{Equation (3)}: & \quad \neg f \leftrightarrow \neg f_a \land \neg f'
\end{align*}
\]

Equation (1) expresses that if the feature \( f \) and \( a \) are selected, the \( f_a \) feature, which links to the code modified by the feature \( a \) must be selected too. Equation (2) ensures that if \( a \) is not selected, but \( f \) is, the feature \( f' \) without the code will be added. Finally, we ensure that, if the feature \( f \) is not selected, neither \( f_a \) nor \( f' \) is selected using Equation (3).

The FeatureChanges(\( a, f \)) relation can be expressed by combining the relationships described in Table I. For clarity, the logical relations as described in Table II are also included.

\[
\begin{align*}
\text{Or(} f, \{ f', f_a \} \text{):} & \quad f \leftrightarrow f' \lor f_a \\
\text{Excludes(} a, f' \text{):} & \quad \neg (a \land f') \\
\text{Requires(} f_a, a \text{):} & \quad f_a \rightarrow a
\end{align*}
\]

It can be proven that these three relations are equivalent to the FeatureChanges relation, which can thus be expressed using these basic relations. This enables us to express this concept in a feature model that supports only a limited amount of relations between features. The feature expansion approach is illustrated in Figure 7, where a feature \( a \) impacts an entire subtree.

It is of note that the newly created features \( f' \) and \( f_a \) are not referenced in the \( D \) feature model, making them part of the set of artificial features as discussed earlier, making them candidates for or elimination.

D. Feature model conversion algorithm

The techniques described above are combined into a feature model conversion algorithm, which we use to convert between development and runtime feature models. The algorithm takes
a development feature model as an input, and first applies feature expansion. The amount of features in the expanded feature model is then reduced using feature elimination, resulting in the runtime feature model.

This feature model conversion algorithm is implemented within the service repository of the architecture discussed in Section IV, where it bridges between the development feature model and the runtime feature model.

VI. EVALUATION RESULTS

We evaluated the algorithm to convert runtime feature models to runtime feature models using three feature models of applications used in the CUSTOMSS project. These applications are in the field of Document Processing (DP), Medical Information Management (MIM) and Medical Communications (MC).

Figure 8 shows the feature models of a MC application, in the various stages of the algorithm. The feature names have been replaced by numbers, where every feature is represented using the same number throughout the different images.

Figure 8a shows the development feature model of the application. For this model, Feature 21 impacts Feature 6, 9, 12, 15, and 18. This is noticeable after expansion, as shown in Figure 8b, where these features have new children, related to Feature 21 using Excludes and Required relations. Finally, after the elimination stage, some features are removed, lowering the total depth of the feature model, as shown in Figure 8c.

Feature expansion increases the amount of features, while feature elimination again decreases the present features in the runtime model. This is shown in Figure 9, where the feature counts for the three applications after the different phases are shown. We make a distinction between empty features, filled features and a total feature count.

Filled features are features linked to code and configuration, and are thus required during the deployment process. Empty features are only used to structure the feature model, and yield overhead during the placement. The increase of the amount of filled features during the feature expansion phase can not be prevented, and is indicative of why the transformation is useful: otherwise this variability needs to be managed during development. The increase in empty features is however not desirable, but as seen in the bar chart, this overhead significantly diminishes due to the feature elimination stage.

It is of note that the most significant increases in feature counts occur in the Document Processing application, where some features are impacted by multiple features, exponentially increasing the amount of variations. Most of the excess empty features created in this process are again removed in the feature elimination process. The exponential increase in features does however indicate a need for limiting the use of the discussed approach to those features that can not be implemented using alternative approaches, and maximally using configuration-based approaches for handling variability.

The execution speeds of the different algorithm phases were determined for the different application feature models. The results, shown in table Table IV, were determined using 1000
executions on an Ubuntu server with Intel Core 2 Duo T9400 CPU and 4GiB of memory. The results show that both algorithm phases can be executed fast, the actual execution speed is mainly influenced by the number of features generated.

To evaluate the impact of the transformation on the management complexity, we used the expanded and runtime feature models as input for the feature placement algorithm discussed in [4]. We considered a scenario containing 100 applications and servers. As seen in Table V, where average times and percentiles are shown for 100 executions, placement consistently executes faster for the runtime models. For the MC and MIM cases the execution duration decreases by ±5%, while for the DP model a reduction by ±17% is observed.

VII. CONCLUSIONS

We presented a software architecture for development, execution and management of highly customizable SaaS applications, and described how SPLE techniques and feature modeling can be utilized within this framework, focusing on how feature models are used at different stages in the software life cycle. The proposed feature model conversion algorithm can be utilized to convert development feature models to runtime feature models, increasing maintainability and expressiveness of development feature models. The transformation can be executed fast, in the order of a few milliseconds, and techniques are proposed to significantly limit the overhead of empty features that is added during the transformation, improving the execution speed of placement algorithms by 5 to 17% depending on the application case.

The proposed approach does come at a cost when features exist that are impacted by many different features, as an exponential amount of feature variations has to be created. This can however be remedied by only using this approach when it is impossible to resolve the variability using other techniques, such as configuration changes or AOP techniques.

In future work, we will extend the execution platform, which uses the feature model conversion and feature placement algorithms, and develop a process management layer enabling QoS aware composition of services.

ACKNOWLEDGEMENT

Hendrik Moens is funded by the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT). This research is partly funded by the IBBT CUSTOMSS[5] project.

REFERENCES


