Automated Constraint-based Multi-tenant SaaS Configuration Support Using XML Filtering Techniques

Azadeh Etedali, Chung-Horng Lung, Samuel Ajila, Igor Veselinovic
Department of Systems and Computer Engineering
Carleton University
Ottawa, Canada

Abstract—The use of cloud computing is on the rise because of the cost effectiveness in providing the same resources to different tenants. Highly customizable Software-as-a-Service (SaaS) provide high scalability and lower cost as a result of multi-tenant nature of its cloud applications. However, there are several challenges that make its adoption difficult. Some of the weaknesses of current cloud offerings are complex configuration, development, deployment and management. This paper, investigates the different techniques and methods such as adaptation and variation management used to address these challenges. In addition, the details about how techniques from Software Product Line Engineering and Service Oriented Architecture are applied are addressed. Therefore, the paper extends an existing framework for constraint-based configuration management for cloud applications by integrating existing feature modeling tools with a XML filtering tool —Yfilter. The objective of the integration is to automate the process to identify matched tenant-specific requirements with the SaaS cloud application feature model. The automated process results in lower manual efforts and possible errors, and as a result less complex deployment and management of the SaaS application.

Keywords—Cloud Computing; Software-as-a-Service; SPLE; Feature Modeling; XML Filtering; Yfilter

I. INTRODUCTION

The interest in cloud computing is rising in recent years. It offers on-demand applications and services to multiple end-users. There are three models of cloud services under consideration, Software as a Service (SaaS), Platform as a Service (PaaS), and Infrastructure as a Service (IaaS). SaaS can be offered to clients at a lower cost as it is usually multi-tenant in the sense that many end-users, even from different organizations, use a single application instance.

In order to explore economies of scale, SaaS providers must be successful in attracting a considerable number of tenants for their SaaS applications. But there are several challenges that should be considered: users and preferences may not be known in advance and may change over time. Even if they are known and limited, different tenants usually have different and even conflicting preferences. This however has the effect of limiting application configurability and customizability [3]. There should be trade-off between providing tenant specific customization and deployment while retaining enough commonalities to exploit economies of scale [1].

Offering the best user experience while considering the changing and conflicting requirements and preferences of tenants, some approaches [4, 8, 9, 10] employ adaptation techniques to make changes in application based on the various existing preferences and configuration options of the applications. Other approaches [1, 2, 3, 6, 7] use variability management techniques to provide different variations based on different tenants’ preferences. They focus on different methods which results in having less variation instances and making most reuse of the currently deployed variations as much as possible.

Using Software Product Line Engineering (SPLE), an application is modeled as a collection of features and relations between these features [7]. Feature modeling helps in reasoning on possible variations of the system [11]. Then, the development phase begins. Service Oriented Architecture (SOA) is a well-suited architecture in this phase. In this case, the individual services map to the different features defined in the feature model. Deploying the application then comes down to allocating feature instances and connecting them [7].

Deployment of software systems and components on clouds can be performed using different approaches: (1) coarse-grained partitioning over different clouds according to the requirements; (2) application of software product line (SPL) technologies to support different variants and configurations; and (3) consideration for data placement, cost, and other non-functional constraints [12]. In [13] a reference framework is defined for constraint-based multi-tenant approach for SaaS development, deployment, and management. Yfilter tool was applied for filtering and matching tenant-specific requirements to configuration options of the SaaS application. Yfilter helps in selecting appropriate cloud configuration or specific service products from a large set of viable options. But since Yfilter accepts specific XPath representation as input, some parts of the work for transferring feature model representation to an acceptable XPath representation, is done manually.

The objective of this paper is to automate the manual process in [13]. An automated process can save time and efforts, and make the management and deployment process easier. We have filled this gap and provided a tool to integrate
feature modeling tools with Yfilter, to take better advantage of existing tools in the target problem.

The remainder of this paper is structured as follows. Section II presents a background of the research area and Section III describes the related works. Section IV reviews the application of Yfilter to the problem and how the provided tool helps in automating the process. Finally, a conclusion is given in Section V.

II. BACKGROUND

A. Multitenancy

Multi-tenancy allows cloud providers to deliver the same service to different customers who share physical and/or virtual resources transparently. Depending on the adopted cloud service model, users can share resources at various levels, from hardware resources (e.g., CPU, storage) to software applications [2]. In this context, the term tenant refers to a user group which represents the stake holders of the same organization. In multi-tenant SaaS, a single instance of the software will run on the service provider's infrastructure, and multiple tenants access the same instance. It lets tenants share the same hardware resources, by offering them one shared application and database instance, while allowing them to configure the application to fit their needs as if it runs in a dedicated environment [14]. Multi-tenancy in SaaS is more cost effective and easy to administer, because the SaaS provider handles, updates and runs only a single instance of the software, eliminating the burden of handling multiple and different software releases [4]. As a result, application maintenance becomes easier for the service provider. Secondly, it can increase the utilization of resources. These two factors reduce the overall cost of the application and encourage high degree of share-ability [14].

However, some major objectives [14] for implementing and monitoring a multi-tenant environment are better performance with high availability, efficient service delivery with zero downtime, cost optimization and managing the applications. Beside these, application management is one of the crucial issues required to attain multi-tenancy, particularly with the shared data sets. Multi-tenancy can support different degrees of isolation. The lower the degree of isolation, the bigger the resources and cost savings, but the smaller the configurability. Limited configurability [1] is a major drawback, especially when user preferences are not known in advance.

B. Multi-tenancy deployment patterns

Based on the need for customer-specific adaptation, a SaaS application (or parts thereof) can be realized and deployed using three basic patterns [1]. These patterns are:

- **Single instance**: denotes that all tenants use the same instance of the SaaS application (or parts thereof), i.e., the instance is shared between all SaaS applications (commonality). The “same instance” means that the same workflow using the same code on the same infra-structure by several tenants.

- **Single configurable instance**: all tenants use the same instance of the SaaS application (or parts thereof). However, the instance is configured on a per-tenant basis. This means that whenever the SaaS application is invoked by the tenant, the instance is adapted during run-time (typically through configuration metadata, e.g., configuration files or configuration entries in a database). This means that there is variability in the SaaS applications.

- **Multiple instances**: each tenant uses a different instance of a service. This allows for the most flexible adaptation to the customer requirements, but also requires separate code to be deployed for each tenant.

C. Software Product Lines and Feature Model

An SPL is a family of software systems developed from a set of common core assets. An SPL creates its products typically before runtime. There are two main ways to implement an SPL: the annotative approach and the compositional approach [5]. The former annotates the features of the SPL in the application code using explicit or implicit annotations, supporting fine-grained variability, but reducing flexibility and maintainability. The latter realizes the features as modular units and creates products by composing these modular units. The compositional approach can potentially reduce the drawbacks of the annotative approach.

In contrast, a Dynamic SPL (DSPL) [15] creates the products at runtime. In general, a DSPL can realize a system/product that adapts its behavior over time based on the systematically engineered variability.

A feature model [5] captures the commonality and variability in an application at a high-level of abstraction. In an SPL, it supports activities such as asset development, and product creation and adaptation. By selecting the features of a feature model respecting its constraints, a feature configuration is created.

D. SOA

The service oriented Architecture (SOA) is well-suited as an architectural style for realizing SaaS applications as it offers a flexible way for building new composite applications out of existing building blocks (i.e. services) [1]. SOA helps realizing SaaS applications quickly, through services reuse, and decreasing the applications time to market and leveraging the economy of scale [2]. According to the SOA paradigm, the development focus shifts from activities concerning the in-house custom design and implementation of the system components, to activities concerning the identification, selection, and composition of services offered by third parties. [16]

E. Variability Management

Software variability is the ability of a software system or artifact to be efficiently extended, changed, customized, or configured for use in a particular context [17]. Different methods are introduced and used for variability modeling, implementation, realization and representation. Variability
management techniques are mainly employed in SPLE area to model, implement and realize different variants of a family of software product.

F. Service-Oriented Dynamic Product Line Architecture

SOAs have proven cost-effective in developing flexible and dynamic software systems. The loose coupling in SOAs can provide DSPLs with the technical underpinnings of flexible feature management. So, converging SOA and DSPL helps developing self-adaptive SOA systems. DSPLs, in turn, can provide the modeling framework to understand a self-adaptive SOA-based system by highlighting the relationships among its parts. Software system architects can use these models to understand the implications of modifying a system’s configuration at runtime [18].

By converging these two techniques, SO-DSPL was proposed based on Single Instance model, to provide the capability of: (1) defining multiple products or service compositions on the same set of shared service assets, and (2) having a runtime representation so that the architecture can be changed dynamically at runtime. To do so, it exploits a compositional technique, where a collaboration among a subset of component services at runtime is used as the composition unit. This can reduce the complexity and rigidity of the application resulting from the embedded variations in the application code. Furthermore, this approach takes advantage of a feature-based approach for creating and reconfiguring application variants used for providing high-level support for tenants [5].

III. RELATED WORK ON MULTI-TENANT SAAS

There are significant research efforts in multi-tenant SaaS to address the existing challenges. Generally, they use a combination of different concepts mentioned in Section II and use different methods in each area to propose a new solution for existing challenges. Here we categorize some of the recent works done, based on their approach.

A. Variation based approach

Most researches in multi-tenant SaaS focus on handling variations of application to support different tenants’ preferences. They usually use techniques and concepts from SPLE. Some researchers use SOA as the architecture of their target systems and map concepts of SPLE to this architecture and try to manage different variations of system using SOA specifications.

1) Variation based on SPLE

In this approach SPLE and variation concepts are applied but not limited to SOA architecture. Ref [3] proposes different forms of customization in development phase and presents a methodology to manage these applications in cloud datacenters and network environments. The work in [1] uses external and internal variability concepts from SPLE and model variability using Orthogonal Variability Model (OVM). Then they generate customization flows out of variability information to guide tenants through customization steps for binding external variation points. The internal variation points are bound by provisioning infrastructure, considering the cost of the binding and the deployment pattern. A deployment script is generated using the annotation of variability model with deployment information. Their solution can be used for all deployment patterns.

2) Variation based on SPLE and SOA

The authors in [2] support application variability on a per-tenant basis and use the concepts of SPLE (variation points, OVM language) to handle variability modeling and management. They focus on single configurable instance deployment pattern and use SOA as software architecture and handle variability management through SOA specification. So, in their solution, a variability management mechanism should be made available for each application layer separately, due to differences between the layers and concepts. The work of [7] focuses on single configurable instance deployment model and applies feature modeling from SPLE to model variability and SOA for system architecture, and proposes methods and algorithms for allocating the different service instances, taking into consideration the distinction between internal and external variabilities in these algorithms. In work presented in [6], they extended their solution by proposing a runtime version of feature model which is used as an input of feature placement algorithm and is used in deployment and provisioning step, and proposes feature model conversion algorithm to transfer from logical feature model to runtime version. Ref [5] focuses on single instance deployment pattern and use DSPL to realize a SOA based application (SO-DSPL) and enhance it with runtime sharing and variation using a compositional technique, where a collaboration among a subset of component services at runtime is used as the composition unit (the reuse unit), instead of the annotative one. A feature-based approach is used for creating and reconfiguring application variants. The work in [19] models the hierarchical workflow template design of SOA systems by OVM, and introduces and uses an ontology based specification to define variation points which assists mining relationships between tenants’ customization decisions and uses this knowledge to automate the customization for future tenants. This solution can be applied to applications based on single configurable instance deployment model.

B. Adaptation based approaches

Instead of considering different variations of an application for different tenants, some researchers try to use adaptation and find out how an application can adapt to the new requirements of different tenants.

The authors of [9] and [10] propose a solution based on Single Instance deployment pattern and SOA applications. A long-running single instance should be upgraded and changed based on the requirements of the tenants and providers. They provide a middleware for runtime adaptation by means of gradual tenant-by-tenant activation of upgrades, maintaining availability and QoS. They extend their work to tenant-aware self-adaptation of the single instance of SaaS application [8].
The work in [4] provides a general solution based on Single Instance deployment pattern. The configuration space is modeled using feature model and represented as FaMA plaintext. Then they use the activities of MAPE loop (as a reference control model for autonomic and self-adaptive systems) to support user-centric adaptation. Proposing an optimization problem which is solved using genetic algorithms, per each change in tenant preferences, the system chooses a configuration option which maximizes the satisfaction of tenants.

C. Discussion

Generally, the adaptation based approaches are built on the Single Instance model. Those approaches intend to satisfy the needs of all tenants. But because of different and conflicting requirements, they can only satisfy the tenants to some level. On the other hand, the variation based approaches are proposed to limit the number of application instances. However, because of different service-layer agreements (SLAs) and workflow for different tenants, the number of variations can grow rapidly.

In comparison to the approaches using the formal method, the approach in [13] has much smaller configuration space. Because the common paths for features in a feature model are shared using Yfilter, instead of enumerating all possible product options. The processing time is also very small, which is suitable for potentially large numbers of features and tenants. But the problem with that work is the amount of time and efforts required to model the application features, because the process of feature modeling is done manually. The manual generation of feature model for each tenant is time consuming and is prone to errors for applications with a large number of features.

IV. MATCHING USING CONSTRAINT-BASED CLOUD DEPLOYMENT FRAMEWORK

A. Constraint-based Cloud Matching Deployment Framework

A great deal of research efforts has addressed the multi-tenancy challenges in SaaS. They are faced with the complex problems of SaaS development, deployment and management, which are due in part to the different variations or complicated processes for adaptation of the single instance. The configuration of these applications, especially in the case of the existence of tenant specific variants, may not satisfy scalability and flexibility. The authors of [13] present a conceptual constraint-based multi-tenant approach for cloud deployment that helps in configuration and management of tenant-specific variants. It uses the cloud services/products configuration options and different tenants’ requirements in the process of managing these variants. It contains four main processes. The framework is presented in Figure 1.

During the process of Tenant Requirements Analysis and Configuration Management (bottom left in Figure 1), the tenant requirements (both functional and non-functional) and constraints are identified and analyzed. These requirements and constraints are gathered from tenants, through the SaaS provider’s configuration tool. Then, the requirements and constraints are processed through the activity Feature Modeling. The tenant-specific features are later encoded in XML format.

The Cloud Services Analysis and Configuration Management (top left in Fig. 1) models cloud services/products using feature models. The feature models include functional and non-functional features, tenant-independent software configurations, and cost constraints. The cloud-specific feature models and the tenant-specific XML documents (obtained from the previous process) are the inputs to the activity Matching Engine which uses Yfilter to select the appropriate cloud service products that match the tenant-specific requirements.

The proposed approach in [13] consists of the following steps for matching tenant requirements with cloud-specific requirements.

- Model the cloud service/product features with feature models and encode it manually in the form of DTD and convert DTD representations to XPath representations
- Model the tenant-specific requirements with feature models and encode it with XML format
- Apply the Yfilter matching technique to identify the matched system features based on the tenant-specific requirements and constraints

Figure 1. Constraint-based Cloud Deployment Framework [13]

The matching engine applies an NFA-based approach, by using Yfilter tool, which has the benefit of tremendous reduction in the space size. This is because the common paths for features in a feature model are shared using Yfilter, instead of enumerating all possible product options. The processing time is also very small, which is suitable for potentially large numbers of features and tenants.

XPath [24] is a language for retrieving, selecting, and filtering information from an XML document. XML filtering is an established technique that has been thoroughly investigated to match an XML document d against a number of XPath feature (query) representations F to identify whether d has the isomorphic tree structure and values defined in F [13]. The Yfilter tool used in this approach, takes queries with the XPath format. Hence, to be able to use this tool, the authors in [13] extended the tool to parse the DTD representation of the service/product feature model and convert it to the required XPath format. But still, encoding the
V. AUTOMATED ENCODING IN CONSTRAINT-BASED CLOUD DEPLOYMENT FRAMEWORK

As stated at the end of section IV, the approach presented in [13] to convert feature models to the desired XPath format for Yfilter, is a manual process. This section presents how the manual process has been automated.

A variety of tools for creating feature models are available for use in software product line development. Some examples include FeatureIDE, an Eclipse-based IDE [20], Software Product Line Online Tools (S.P.L.O.T.), an online set of feature modeling tools [21], and BeTTy [22] which provides an online feature model generator. This research focuses on providing a method to take the feature models produced by these tools and to produce XPath representations of the features in order to make use of Yfilter [23], which was used in [13].

FeatureIDE and S.P.L.O.T. provide the ability to produce fully user-created feature models and export them in the Simple XML Feature Model format (SXFM) [20, 21]. BeTTy provides an online tool for randomly producing sets of feature models based on user-specified constraints and allows these feature models to also be exported in the SXFM format [22]. This consistent option between the three tools was the reason for the decision to develop a tool to translate each feature model in an SXFM to its equivalent XPath representations.

The researchers behind S.P.L.O.T. developed a Java library that allows the parsing of SXFM files [21]. This library is being used to develop a Java application with a graphical user interface which reads SXFM files and produces the corresponding XPath representations in a single text file to be used by Yfilter tool. With the developed Java application, any feature models produced by and exported from FeatureIDE, S.P.L.O.T., or BeTTy can be quickly converted to the XPath format that is required by Yfilter. This approach greatly reduces the time needed to produce this format, as the previous method required the writing of an entire document type definition (DTD) file manually for a single feature model which could then be converted to the desired XPath format by Yfilter [13].

VI. EVALUATION

We have developed an application to integrate the existing feature modeling tools with the Yfilter tool. We have evaluated the work by using the following two scenarios:

Scenario A. Follow the manual process in [13]: This scenario requires the following steps:

1) Assuming that the feature model is constructed manually, model SaaS system features by generating a feature model in DTD format and run a tool supported by Yfilter to convert the feature model to XPath.

2) Model tenants’ preferences in XML format.

3) Run Yfilter to match SaaS system features with tenant’s requirements.

Scenario B. Use our proposed process described in Section V, the following steps are needed for this scenario:

1) Perform SaaS feature modeling using S.P.L.O.T. This is done by creating a graphical feature model and then exporting the feature model to XSFM format using the available tool.

2) Convert the XSFM file to XPath format using our tool.

3) Model tenants’ preferences in XML format.

4) Run Yfilter to match SaaS system features with tenant’s requirements. We used the same aforementioned feature models for a comparison.

We used the same tenant’s requirements in XML for both scenarios. Also, we used the same SaaS system feature models. We modeled two different Systems: (i) Part of Amazon EC2 with 35 features and (ii) Part of Windows 8 with 77 features.

The comparison is based on the following criteria:

1) Pre-Yfilter Time: Pre-Yfilter actions are everything that needs to be done to prepare the queries file (system feature model) and tenant’s preferences XML file for Yfilter. After these files are prepared, Yfilter could be run to start the process of matching. For Scenario A, the Pre-Yfilter time includes the first two steps, but it consists of steps one to three for Scenario B. We check the time taken on step 1 for both scenarios and measure the time spent for automated steps in each scenario.

TABLE I shows that although the queries generation and Yfilter only needs to start the process of matching. In Scenario A, the Pre-Yfilter time includes the first two steps, but it consists of steps one to three for Scenario B. We check the time taken on step 1 for both scenarios and measure the time spent for automated steps in each scenario.

TABLE I shows that using the S.P.L.O.T. tool we could generate the system feature model in less than half of the time spent for manually generating it. Moreover, in generating the DTD model manually in Scenario A, designers have to take care of the syntax and also correctness of the model. There is still a risk of errors in the model, because the existing validation tools do not analyze the DTD model and are only able to validate the XML syntax. On the other hand, the S.P.L.O.T. tool has a feature to analyze the generated model. Reviewing the results gives some idea of possible mistakes during modeling the features. Finally, the S.P.L.O.T. GUI is much easier to use comparing to the text based editors, e.g., eclipse xml editor or Vim, required in Scenario A.

2) Yfilter Response Time: For Scenario A, Yfilter should first convert the DTD model to XPath queries and then start the matching process. So, the response time for Yfilter includes the time required for both steps. In Scenario B, the XPath file is provided directly to Yfilter, and Yfilter only needs to start the process of matching. TABLE I show that although the queries generation takes less than a second, it makes much difference in Yfilter response time when it is compared in both scenarios.
3) Accuracy of results: To ensure the correctness of our approach, we compared the results generated by YFilter for the same application feature models in both scenarios. The comparison shows that both scenarios generate the same result.

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Number of Features</th>
<th>Pre-YFilter Time (min)</th>
<th>Generating Queries by YFilter (ms)</th>
<th>Matching by YFilter (ms)</th>
<th>YFilter Response Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>77</td>
<td>50</td>
<td>88.7</td>
<td>3.1</td>
<td>91.8</td>
</tr>
<tr>
<td>B</td>
<td>77</td>
<td>20</td>
<td>0</td>
<td>3.2</td>
<td>3.2</td>
</tr>
<tr>
<td>A</td>
<td>35</td>
<td>30</td>
<td>71.1</td>
<td>2.6</td>
<td>73.7</td>
</tr>
<tr>
<td>B</td>
<td>35</td>
<td>12</td>
<td>0</td>
<td>2.5</td>
<td>2.5</td>
</tr>
</tbody>
</table>

This paper focuses on the automated process of generating the queries. A comparison of execution time for YFilter and other methods was presented in [13]. Our evaluation shows that the proposed automated approach saves much time and effort to model the application, while generating the same result. According to “Matching by YFilter” time in Table 1, the execution time of the filtering is identical to previous work. If the number of features grows, the amount of time and effort saved using our approach will be higher.

VII. CONCLUSIONS

A great deal of research has been done in the area of SaaS for addressing the challenges of multi-tenancy in order to have best tenant experiences and for reusing the same application. In this paper, we have extended an existing work on matching the system requirements with tenants’ preferences to help deploy the application variant. In our work, we took advantage of existing feature modeling tools to help in modeling system features and integrated it with the YFilter tool used for matching. We automated the process by analyzing the steps and developing a tool to convert the XSFM output of the feature modeling tools to the XPath format required by YFilter. This way we could make an end-to-end solution and replace the manual and error-prone process with an automated one.

We have evaluated the proposed automated process by comparing this approach with the manual one used in [13]. The matching results were identical, while the time spent on preparing the queries using the automated process was only half of that used in [13] assuming that the SaaS feature model has been constructed.

We are currently working on the extension of FeatureIDE to add the capability to export feature models to the XPath format. This allows for the immediate production of the XPath representations of the features in a feature model by the tool where it was first designed.

REFERENCES