Architecting Cloud Tools using Software Product Line Techniques: an Exploratory Study

Leonardo P. Tizzei
IBM Research | Brazil
ltizzei@br.ibm.com

Leonardo G. Azevedo
IBM Research | Brazil
lga@br.ibm.com

Maximilien de Bayser
IBM Research | Brazil
mbayser@br.ibm.com

Renato F. G. Cerqueira
IBM Research | Brazil
rcerq@br.ibm.com

ABSTRACT

Multitenant cloud computing tools are usually complex and have to manage variabilities to support customization. Software Product Line (SPL) techniques have been successfully applied in the industry to manage variability in complex systems. However, few works in the literature discuss the application of SPL techniques to architect industry cloud computing tools, resulting in a lack of support to cloud architects on how to apply such techniques. This work presents how software product line techniques can be applied for architecting cloud tools, and discusses the benefits, drawbacks, and some challenges of applying such techniques to develop a real industry cloud tool, named as Installation Service.

1. INTRODUCTION

Cloud computing is a model for enabling on-demand network access to a shared pool of configurable computing resources that can be rapidly provisioned and released with minimal management effort or service provider interaction [1]. Cloud computing environments usually fall into three service models: (i) Infrastructure-as-a-Service (IaaS) provides to consumers processing, storage, networks, and other fundamental computing resources where consumers are able to deploy and run arbitrary software; (ii) Platform-as-a-Service (PaaS) provides to consumers capabilities to deploy, onto the cloud infrastructure, consumer-created or acquired applications created using programming languages, libraries, services, and tools supported by the provider, although compatible resources coming from other sources can be used; (iii) Software-as-a-Service (SaaS) provides to consumers capabilities to use provider's applications, running on a cloud infrastructure, through either a thin client interface, such as a web browser, or a program interface [1].

Multitenant cloud computing tools support the deployment of multiple clients (*e.g.*, organizations), so these tools should handle variabilities to enable customizations that fit client requirements [2]. For instance, clients of such a tool might require different Database Management System (DBMS), (*e.g.*, DB2 or Oracle) to store their data. On the other hand, the cloud tool may require the capability to be deployed in different IaaS cloud providers, such as SoftLayer and Openstack. Cloud computing tools should resolve these variabilities in order to be configured. Multitenant cloud tools can be complex and they might have to handle variabilities in different layers of the cloud (*e.g.*, IaaS, PaaS, and SaaS) and at different development phases (*e.g.*, requirements, architecture design, implementation) in order to meet the requirements of different clients.

Software Product Line (SPL) techniques have been successfully applied in the industry [3] to manage variability in complex systems. A SPL is a set of software-intensive systems sharing a common, managed set of features [4]. SPL architectures should support software variability [4], which enables the coexistence of several product architectures sharing common elements. Several methods in the literature provide guidelines to design SPL architectures using feature models. These guidelines define a mapping between features and architectural elements, which helps architects to reason how changes in the feature model can affect the SPL architecture elements. A few works in the literature discuss the application of SPL techniques to develop industry cloud computing environments [5]. Most of them focuses on using feature modeling on IaaS layer, so few of these works consider PaaS and SaaS layers. Thus, the problem this work mitigates is that practitioners have little support to apply SPL techniques for *architecting* their cloud tools and, particularly, PaaS and SaaS tools.

The main contributions of this work is twofold. As first contribution, we describe how SPL techniques can be applied for architecting cloud computing tools. In particular, we show how a feature-oriented method, namely FORM [6], can be used to architect a cloud tool by supporting the architect to model variabilities and map them to cloud architectural elements (*i.e.*, components and connectors). We show how SPL techniques can be applied to design architectural variabilities and also how implementation-level variabilities can be implemented within architectural elements.

A second contribution is the discussion about benefits, drawbacks, and challenges of applying SPL techniques to architect a real industry tool (named as Installation Service) for cloud computing environment. Installation Service is a proof-of-concept tool that aims at supporting the installation of software on multiple virtual machines (VMs), *i.e.*, it provides to consumers capabilities to deploy consumer-created or acquired applications in VMs. Thus, we classify it as a PaaS tool, even though it also has some characteristics of SaaS applications according NIST definitions [1], such as being accessed through a web browser or program interface. Software installation is performed on infrastructure, so Installation Service should also consider how to deal with IaaS layer capabilities, which makes the Installation Service implementation more complex.

The remainder of this work is divided as follows. Section 2 briefly introduces basic concepts on SPL. Section 3 presents the Installation Service features and how SPL techniques were applied to its development. Section 4 discuss the benefits and drawbacks of applying SPL techniques and Section 5 presents the challenges found during this investigation. Section 6 presents the related work and Section 7 concludes and presents future work.

2. FUNDAMENTAL CONCEPTS OF SOFTWARE PRODUCT LINE ENGINEERING

"A software product line (SPL) is a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way" [4].

Software variability enables the derivation of different products from a common set of core assets, and it is defined as the capacity of software system to be customized in a particular context [7]. A variation point represents a software variability in domain artifacts. Software variants are variation point alternatives. The resolution of variation points are part of the process called product configuration [8]. Software variability is orthogonal to SPL development [8] and, thus, it should be modeled from early development phases, such as, feature modeling (Section 2.1), through SPL architecture (Section 2.2) up to SPL implementation [7].

In the following sections we describe SPL requirements, architecture design and implementation, focusing on the techniques that we applied on this work.

2.1 Software Product Lines Requirements

The main goal of requirements engineering is the specification of common and variable domain requirements [8]. We applied a requirement engineering technique called *feature modeling*. A feature is a characteristic of a system relevant to stakeholders [9] and it is used to capture SPL common and variable parts. The feature model is a hierarchical decomposition of a system in terms of features and their relationships. Features can be (i) mandatory, which are common to all products; (ii) optional, which may or may not be selected for a particular product; (iii) alternative, which results in one or more features being selected from a set.

Feature-Oriented Reuse Method (FORM) [6] is a feature modeling approach that encompasses activities for identifying and classifying features. Domain experts abstracts domain knowledge by means of feature identification. These features are then classified according to four layers: (i) *Capability* layer describes features that represent services, operations, functional and quality attributes (*i.e.*, non-functional requirements); (ii) *Operating environment* layer describes features that represent attributes of the environment in which an application is used and operated; (iii) *Domain technology* layer encompasses features that represent implementation details of domain technology; and, (iv) *Implementation technique* layer encompasses features that represent also implementation details that are more generic (non-domain specific). After classification, features are modeled by creating a feature diagram, which is a graphical tree-form representation of features and their relationships.

2.2 Software Product Lines Architectures

SPL architecture is a type of software architecture that, according to SEI [10], is a core asset for all products. SPL architecture provides variation mechanisms that support the diversity among products and it is composed by mandatory and variable architectural elements [11].

Product architectures (also known as application architectures) are SPL architecture specializations. The former bind the variability of the latter and introduce product-specific changes according to product requirements [8]. A product architecture corresponds to an architectural configuration, that is, a connected graph of components and connectors that describes architectural structure [12].

FORM [6] also provides guidelines to design SPL architectures, which comprise the following models: (i) *Subsystem model* defines the overall structure by grouping functionalities into subsystems; (ii) *Process model* represents the dynamic behavior of each subsystem; and, (iii) *Module model* relies on features at all levels to identify components. In this work, we used FORM because it supports feature modeling and the transition from features to architectural design.

2.3 Software Product Line Implementation

SPL implementation goal is to build a system according to the SPL architecture [8]. Architectural elements and their interfaces are implemented realizing the variability described in the feature model. The SPL architecture determines most of the variability [8], but some fine-grained variability might be implemented within the architectural elements. That is, instead of having one architectural element realizing a single variable feature, two or more variable features are realized by a single architecture element. The realization defines different binding times of variability [8], such as compile time, load time, and run-time.

3. APPLYING SOFTWARE PRODUCT LINE TECHNIQUES

This section describes how we applied SPL techniques to the development of Installation Service tool.

3.1 Target application: Installation Service tool

A key goal of Installation Service is to facilitate software installation on cloud nodes by inexperienced users in a flexible fashion. For instance, users might select one among different cloud providers (*e.g.*, SoftLayer[1], OpenStack[2]) for VM provisioning and also choose different operating system for his/her VM (*e.g.*, Windows or Linux). Besides, Installation Server supports services to be consumed by other Cloud tools that requires software installation, *e.g.*, tools for nodes provisioning, nodes monitoring, and disaster recovery.

Installation Service brings many benefits for developers and operators that have to replicate similar configuration

[1]`http://www.softlayer.com`
[2]`https://www.openstack.org`

on several nodes. Using this tool, one can easily bootstrap new nodes using either homemade configuration scripts or communities[3] scripts.

The main features supported by the current version of Installation Service are:

- **Bootstrap a node**: installs the required tools on a node (*i.e.*, a physical, virtual, or cloud location) in order to make it able to download configuration scripts from a remote repository and to execute them locally;

- **Update configuration scripts**: supports upload of authored configuration scripts to a remote repository as well as download of configuration scripts from the repository;

- **Assign configuration scripts to a node**: supports association of repository scripts to nodes;

- **Synchronize a node**: runs the most recent configuration scripts in a bootstrapped node.

Installation Service is composed by two modules: `Installation Service API` and `Installation Service UI`. The latter provides a user-friendly web interface to manage software installation and configuration scripts. It consumes the services provided by the `Installation Service API`. `Installation Service API` provides RESTful web services that encapsulate automation infrastructure tools. Hence, `Installation Service API` consumers have a standard interface that works with different automation infrastructure tools.

The current version of Installation Service supports Chef, although the design allows the use of other automation infrastructure tools (*e.g.*, Puppet or SmartFrog). Chef is a systems and cloud automation infrastructure tool to deploy servers and applications into nodes [13]. Configuration scripts are declarative and abstract definitions (known as recipes and cookbooks) written in Ruby[4]. They are managed like source code and are applied automatically. They describe how a specific part of an infrastructure should be built and managed, *e.g.*, how to install and configure MySQL Database Management System (DBMS).

Chef was chosen due to: (i) It is a mature tool with one of the most complete offerings; (ii) It has a large user community that provides a repository of installation scripts; (iii) Installation scripts are written in a declarative way.

3.2 SPL techniques applied to Cloud Requirements

In particular, our feature modeling activity was based on the FORM approach. Besides the standard feature modeling activities (described in Section 2.1), FORM's classification activity seems suitable for cloud computing environments. Feature identification involved domain experts and existing documentation (*e.g.*, requirements, slide decks). Figure 1 presents the feature diagram of Installation Service, and its composition rules, which were omitted due to space reasons. Following the FORM approach, these features were classified in the four layers presented as follows.

[3]`https://supermarket.getchef.com/cookbooks-directory`
[4]`https://www.ruby-lang.org`

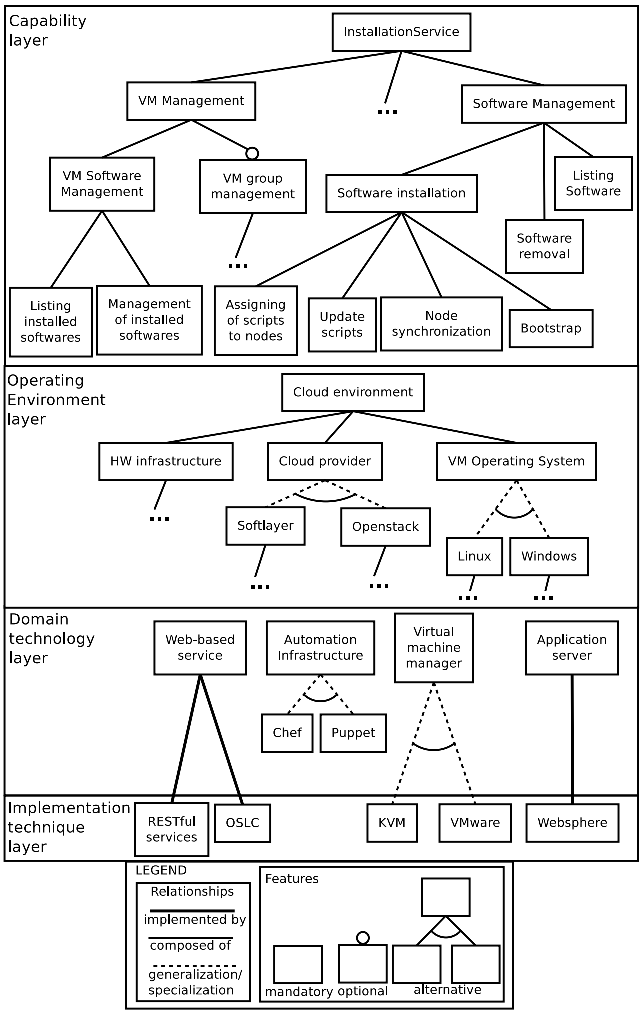

Figure 1: The feature model of Installation Service

- *Capability* layer represents the services provided by Installation Service, *e.g.*, `Installation Service` is composed by the mandatory features `VM Management` and `Software Management`. `VM Management` provides a mandatory `VM Software Management` and an optional `Group VMs` features. `Software Management` provides `Listing installed softwares` and `Management of installed softwares` mandatory features. Such features are closely related to end-users of such tool;

- *Operating environment* layer represents the attributes of the system in which the Installation Service tool is operated. Such attributes of the system can be modeled as features that are well suitable for characterizing the IaaS layer, *e.g.*, in the Installation Service's feature model, `HW Infrastructure`, `Cloud provider` and `VM Operating System` are mandatory attributes of the `Cloud environment`. The `Cloud provider` is specialized in `Softlayer` and `Openstack`, which are alternative features, because these are the cloud providers supported by the Installation Service. Such features does not represent the Installation Service functionalities, but since Installation Service is built on top of IaaS layer, Installation Service depends on these features. For instance, if the cloud provider did not offer support for Windows VMs, the Installation Service would not be able to install software on Windows;

- *Domain technology* layer represents technical details of a particular domain, *e.g.*, `Web-based service`, `Automation infrastructure`, `Virtual machine manager` and `Application server` are Installation Service mandatory technology features. An `Automation Infrastructure` might be alternatively Chef or Puppet;

- *Implementation technique* layer represents technical details that have a generic scope. Example of technologies that can be employed by the tool are `RESTful services`, `OSLC`, `KVM`, `VMWare` and `Websphere`. The `Web-based service` is implemented by `RESTful services` and `OSLC`. Open Services for Lifecycle Collaboration (OSLC)[5] is an initiative to define specifications that enable easier and more effective integration among tools. OSLC specifications allow conforming independent software and product lifecycle tools to integrate their data and workflows in support to end-to-end lifecycle processes.

3.3 SPL techniques applied to Cloud Architectural Design

We followed FORM [6] guidelines for creating structural and behavioral models in order to design Installation Service architecture. We use components to model the SPL architecture, because they contribute to the modularization and reuse [14]. Besides, they can also realize architectural variability by means of two techniques: *adaptation* and *extension* techniques.

In the adaptation technique, a single implementation of a given component is available, but it provides interfaces to adjust its behavior [3]. For instance, in Figure 2, which is based on Gomaa's notation [11] for representing variant (*i.e.*, optional or alternative) and common (*i.e.*, mandatory) components, the `InstallationServiceRestServices` component provides services independent of automation infrastructure for node management (*e.g.*, node creation, removal) and software management (*e.g.*, software installation, software removal). It can work with both Chef and Puppet automation infrastructures, represented by `ChefAdapter` and `PuppetAdapter` variants, respectively. A parameter in the REST API adjusts the behavior of `InstallationServiceRestServices` component defining whether Chef or Puppet should be used. This parameter can be defined by either users using the Web-based portal or other tools that consumes the `InstallationService API`.

The extension technique requires that some components provide interfaces that allow adding new components to it [3]. For instance, in Figure 2, the `SoftlayerAdapter` component provides services that can be consumed by `ChefAdapter` and `PuppetAdapter` components. If we added a new component in the architecture, *e.g.*, a `SmartFrogAdapter`, it could also use the `SoftLayerAdapter` capabilities and it would probably not be necessary to change its implementation.

3.4 SPL techniques applied to Cloud Implementation

[5]`http://open-services.net/`

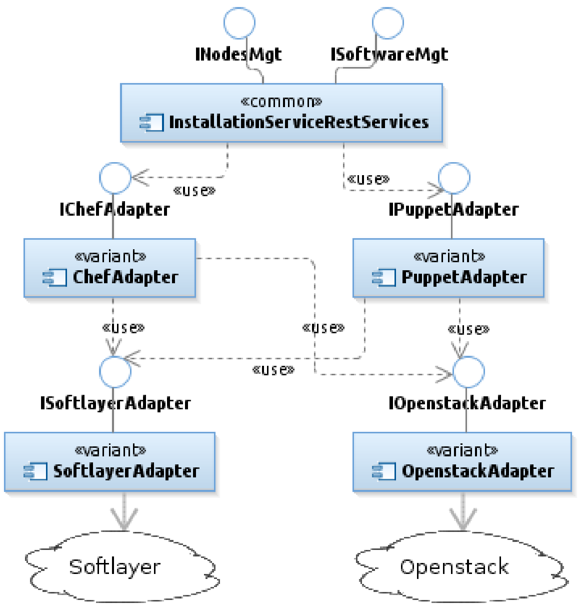

Figure 2: Component-based software architecture of the Installation Service tool

We have implemented fine-grained variability with *parameterization* technique [15] [3]. In this technique, a parameter of a method defines its behavior so it can implement one feature or another. For instance, the Installation Service tool should support nodes with different operating systems, such as Windows and Linux. The implementation of the bootstrap feature is slightly different for each operating system. Figure 3 shows how variability was implemented. The `bootstrap` method has the `opsys` parameter. It has a common implementation and also a specific implementation for each operating system, which is highlighted from line 4 to line 6. This is a fine-grained version of the adaptation technique mentioned in Section 3.3.

```
1  public Response bootstrap(.., String opsys)
2  { //common implementation
3     switch(opsys){
4       case WINDOWS: //do something
5       case UBUNTU: //do another thing
6       case CENTOS: //do yet another thing
7     }// continue common implementation
8  }
```

Figure 3: A Java example of implementation of the parameterization technique

Another variability related to operating systems is that the Installation Service should run on either Windows or Linux, *i.e.*, it is related to the operating system in which the tool runs and not to the operating systems installed on nodes. To resolve this variability, we used the *adaptation* technique [3] by means of a configuration file that was created for each operating system. This configuration file defines the proper values for variable attributes of the class that handles different operating systems. In contrast to the previous variability examples, this one is resolved at compile-time by placing the correct configuration file.

4. DISCUSSION

This section discusses the benefits and drawbacks of applying SPL techniques to cloud development focusing on the exemplary use to develop Installation Service (Section 3).

4.1 Separation of concerns and Multitenancy

The classification step proposed by FORM approach in four layers is not a standard step in feature modeling. However, it helped to manage features that are related to different types of information regarding the four FORM layers. Feature modeling and configuration involve different stakeholders, so FORM layers can support cloud stakeholders to identify which features they should concern with. Tenants are interested in *capability* features, while domain analysts and developers specify *domain technology* and *implementation technique* features, respectively.

The *operating environment* layer of FORM is suitable for modeling IaaS features, because it "represents the environment in which the application is operated" [6]. For instance, typical IaaS cloud providers enable their clients to create VMs by selecting: number of processors *per* VM, number of cores *per* processor, memory, disk, operating system, and so on. This does not mean that all features that model the IaaS are on the *operating environment* layer, but it certainly models most of them.

Features that model PaaS are scattered over *capability*, *domain technology*, and *implementation technique* layers. For instance, tools available on PaaS layer (*e.g.*, Chef and Puppet) are modeled as *domain technology* features, while functionalities of the PaaS tool under development (*e.g.*, Installation Service) are modeled as *capability* features.

Although it is likely that some stakeholders would be interested in more than one layer, the separation of concerns suggested in FORM approach helps the stakeholders to focus on their area of interest. This can be useful as feature models of cloud computing environments are often large and complex. FORM's layers can also support architects to group features that should be implemented by a particular component, because the layers provide a logical separation that should ideally be represented in the cloud tools architecture.

Furthermore, the mapping between features and architectural elements suggested by FORM supports the propagation of change requests from requirements (in this case, modeled as features) to the architecture. Suppose the architecture has a component that provides communication with a particular IaaS cloud provider, such as `SoftLayerAdapter` in Figure 2. If SoftLayer decides to stop supporting Windows VMs, one can easily identifies that it impacts the *operating environment* layer and the `Windows` feature in Figure 1 would be removed. Besides, `SoftLayerAdapter` would be modified to address this change.

In addition, SaaS and PaaS can also benefit from FORM classification, because some features are common to all cloud computing environments, such as services accessible via web browser or web services API [16].

The configuration of a feature model of a multitenant cloud computing environment does not represent a product, but the set of features available for a tenant. Moreover, the configuration process might be performed in different stages, as a tenant pays for a subset of features and within this subset the user might choose a particular set of features at runtime. Schroeter *et al.* [17] applied such *staged configuration* to cloud development and claimed that it is suitable for this context.

4.2 SLA specification

Service level agreement (SLA) is a key aspect for cloud computing. Feature modeling can be applied to reduce the complexity of SLA [18] and the four FORM layers may contribute to this. SLA features related to IaaS, such as geolocation of the data-center, are likely to be represented by features in the *operating environment* layer (see Section 4.1). SLA features related to cloud tools that operate on PaaS, such as security and availability of cloud tools, are likely to be represented on other FORM layers. Algorithms for cryptography that implement security feature can be represented as features in the *implementation technique* layer, for example. Particularly, features such as security and availability are properties of dependability and they rely on dependable architectures to be addressed [19]. Therefore, feature modeling approaches that also provide guidelines to map features to architectural elements can reduce the complexity of SLA and also help to achieve such quality attributes by supporting their architectural design.

Although the feature model supports cloud requirements and architecture specification, there is still room for providing a better support for cloud stakeholders (*e.g.*, domain analysts and users). For instance, an extension of the feature model to represent cardinalities [9] might be useful to represent the number of available nodes for a particular tenant. Feature attributes is another extension that enable the representation of information, such as, pricing, availability, and elasticity that are related to a particular feature [17]. One approach is to extend domain feature model (EMF) with attributes to express this kind of feature variability [20]. Czarnecki *et al.* point modeling attributes allowing a feature to be associated with a type, such as integer or string, as proposed by Bednasch [21]. Attributes can be modeled as subfeatures, each one associated to a type.

4.3 Feature dependencies management

As well as occurs to non-cloud computing environments, having optional and alternative features may cause inconsistencies, because the implementation of a feature may depend on the implementation of another one, and not every feature implementation is available in all configurations [22]. For instance, a mandatory feature that depends on an optional feature causes inconsistencies because the optional feature is not available for all configurations. Such mistakes might happen as cloud computing environments are often complex, having several features and complex interactions among them. Thus, architects should be careful to detect these inconsistencies and, ideally, they should have support of tools for these analyses, because features and their variabilities can be employed to identify architectural elements and their relations.

4.4 Component-based representation

The requirements of the cloud computing environment that comprises the Installation Service tool specifies that its internal modules should be independent and also able to communicate with each other. They also should enable the customization for a particular tenant. These requirements influenced our decision of implementing the tool using components. Furthermore, some of the cloud architecture variabilities should be resolved at runtime. For instance, a user might select SoftLayer as cloud provider because it offers the cheapest computing resources given a certain infrastructure configuration. Thus, we applied two composition techniques that are suitable for this context, namely extension and adaptation. Other works in the literature (*e.g.*, Walraven *et al.* [23] and Ruehl and Andelfinger [24]) also used components to represent SPL architectural elements and as a composition technique to realize architectural variability. Components support the modularization of feature implementation [14] and they are also suitable for composition techniques that support runtime variability. However, to the best of our knowledge, there is no component-based development method specific for cloud computing.

Furthermore, feature modeling supports the architect to identify which components are mandatory (*i.e.*, they are used by all tenants) and which ones are optional and alternatives (*i.e.*, they are *not* used by all tenants).

4.5 Reuse identification

The feature model supports the identification of reuse opportunities [6]. It helped to identify common features for all tenants and the ones realized by mandatory components. Furthermore, cloud computing environments are usually large and complex systems that should support customizations for particular tenants [17] [23]. The feature model contributes to manage such complexity and supports customization by representing feature variability that corresponds to tenant-specific configurations.

5. CHALLENGES

5.1 Feature model to support users

As cloud computing environments are becoming more widespread, inexperienced users are increasingly being pushed to deploy applications in the cloud. For instance, some companies that develop scientific software applications, such as oil reservoirs simulators, are deploying their applications in the cloud to outsource the maintenance of expensive data centers[6]. Scientific software end-users often also assume the developer role in order to address their own scientific problems [25]. Thus, the deployment of applications in the cloud computing environment might be a daunting task for these user-developers without experience on cloud setup. On the one hand, if they require more computing resources than necessary by the application, costs will increase, but on the other hand, less computing resources might delay the results or even crash the application due to lack of memory. Therefore, obtaining the right balance of configuration options might be difficult or time consuming for these users.

The feature model might facilitate the cloud computing environment configuration by inexperienced users since a feature is a high-level concept that is easier to understand than language-specific scripts. To the best of our knowledge, most of the related work presented in Section 6 that uses or extends the feature model for cloud development does not support a semi-automated deployment of applications in cloud computing environments. That is, a tool that analyzes the selected features to recommend cloud computing environment configurations based on rules. For instance, two different scientific methods are represented by alternative features. One method requires more intensive network communication among the nodes, while the other requires more processor intensive nodes. This tool would suggest the best resource allocation given the selected feature. Furthermore, the configuration process is not necessarily based on an explicit feature diagram, but it can benefit from a more user-friendly interface such as a Web form.

Another issue is that cloud computing environments are often complex so one might expect large feature models to represent them. It might be difficult to select proper configuration of large feature models. Thus, automated reasoning tools can guarantee the correctness of a particular configuration and also automate the selection of features in order to optimize non-functional requirements [23]. *E.g.*, Benavides *et al.* [22] mapped the feature model onto a constraint satisfaction problem thus enabling to automatically find an optimal solution defined in terms of one or more variables.

5.2 Cloud architecture variability modeling

A requirement for the Installation Service architecture is the ability to be customized [2]. The implementation of this requirement can benefit from existing SPL architecture techniques, but there are a few approaches in the literature on the integration of cloud computing and SPL techniques [5] for architecture modeling.

Several works in the literature combine Service-Oriented Architecture (SOA) and dynamic SPL (DSPL) [26], which handles product reconfiguration at runtime. Some of these work may be adapted to support cloud computing, particularly SaaS, as it is realized by SOA [27]. For instance, approaches like Kumara *et al.* [28] are based on DSPL techniques to model and implement cloud computing environments. However, most of them focuses on IaaS layer and they do not provide guidelines for architecting cloud tools using SPL techniques.

As cloud architectures for multitenant applications are generally more sound [16], the Common Variability Language (CVL) [29], which supports the specification of variability and resolution models, can support the application of DSPL techniques to design cloud architectures. Besides customization, CVL can also support architectural specification of fault-tolerance [2] requirement. Different fault-tolerance techniques (*e.g.*, recovery block, N-version programming) can be alternative features that are suitable for a particular context and are defined at runtime. For instance, Nascimento *et al.* [30] use CVL to define a fault-tolerant technique more adapted to the service context, but their work is not focused on cloud tools.

6. RELATED WORK

Walraven *et al.* [23] proposed a method for multitenant SaaS applications based on service line engineering. This method is feature-oriented and supports tenant-specific configurations in addition to runtime resolution of variation points. Some techniques described in our study are part of their method, such as feature-oriented support for tenant-specific configurations and composition techniques to resolve variability, but while they described a new method, we applied existing techniques to cloud development.

Ruehl and Andelfinger [24] presented a generic model to

[6]`http://scienceclouds.org/`

allow the creation of flexible SaaS applications. Then, they described how SPL approaches, such as SPL processes, variability description and variability realization, can be applied based on this generic model. Their conclusion is that SPL approaches can be applied to SaaS applications, although some adaptations or extensions might be necessary. Our study complements the works of Walraven *et al.* and Ruehl and Andelfinger by presenting a more low-level description of implementation techniques.

Schroeter *et al.* [17] identified the requirements for a configuration process for cloud computing environments. Based on these requirements they proposed a dynamic configuration process that supports multitenancy. Whereas their approach is focused on the feature model and configuration, our study describes the application of existing techniques to different cloud development phases including architecture design and implementation.

Mietzner [31] proposed a method and implementation for cloud development. His contributions encompass meta-models to support variability modeling, component-based architecture definitions and tools that manipulate the meta-models. Our study complements his comprehensive work by providing a new point of view of how existing SPL techniques, such as FORM, can support cloud development.

Kumara *et al.* [28] proposed an approach to realize service-based single-instance multitenant applications. Their approach supports runtime sharing and variation across tenants and it is based on DSPL techniques. They have also developed a prototype tool and evaluate their approach concerning performance, development effort, and sharing (*i.e.*, reuse of code compared to multi-instance multitenant approaches). Our study is complementary to their approach as we described how compositional techniques can be applied to cloud architecture and implementation.

Schmid and Rummler [5] discuss the benefits and drawback in applying SPL techniques to cloud applications. They describe an hypothetical business information system and infrastructure and how it could be customized with respect to distinct requirement dimensions such as user interface, deployment, software stack, third-party providers services. These customizations are discussed from the point of view of three layers: IaaS, PaaS, and SaaS. They concluded that main SPL approaches lead to different implementation thus increasing maintenance efforts. Another conclusion is that few SPL approaches support runtime resource sharing and enable complexity reduction. In contrast to their work, we discuss based on real industry cloud tool and we take low level details into account.

Wittern *et al.* [32] extended the feature model by introducing feature attributes, feature types, and model types to support for cloud configuration. The model types are further subdivided in three models based on features: (i) *Domain model* represents all relevant abstract decision aspects of the selection problem; (ii) *Service model* represents a single concrete cloud service; and, (iii) *Requirements model* represents the requirements that a stakeholder has with respect to abstract decision aspects. They also described a cloud service selection process that involves the creation of these three models. In contrast to their approach, we applied existing techniques in different phases of cloud development.

Quinton *et al.* [33] proposed an approach to support the configuration of multiple clouds by means of features. They extended the feature model with feature cardinalities and feature attributes. In their work, each feature model represents a cloud provider. They also consider that the user might select features from different cloud providers, that is, a multi-cloud environment. Thus, they use a ontology to map the concepts (*i.e.*, features) of distinct cloud providers. Based on these extended feature models and ontologies, they built a tool to support the selection of only valid cloud configurations. Our approach focusing on a single SPL and provide implementation details, as opposed to their work.

7. CONCLUSION AND FUTURE WORK

SPL techniques can support the architectural design of cloud computing tools, but a few works describe how these techniques can be applied in different phases of cloud development. The main contribution of this work is to apply SPL techniques in a real industry cloud tool to support the transition from requirements to architectural design, modeling architecture, and its implementation. This exploratory study increases the confidence of practitioners to apply SPL techniques to architect cloud computing tools, since it discusses how these techniques can be applied and also their benefits, drawbacks, and challenges.

Cloud tools handle their own variabilities to support multitenancy, they also handle variabilities of the layer in which it is operated (in this case, Installation Service operates on PaaS), and they handle variabilities of layers that they depend on (*i.e.*, IaaS). Feature modeling helps the architect to model such variabilities and map them to the architecture. In particular, the usage of FORM layers leveraged separation of concerns, which supports the architectural design. Existing SPL techniques, such as adaptation and extension, can support modeling architectural variabilities of cloud tools. Otherwise, such variabilities would be modeled and implemented in an *ad hoc* manner.

As future work we are going to apply SPL techniques to architect the entire cloud computing environment and to conduct a rigorous evaluation on how such techniques support reuse and modularization. Another topic of studies is self-adaptive system techniques to support rapid-adaptation of cloud requirements according to user demands and the current use of the cloud resources. For example, Schroeter *et al.* [34] propose a self-adaptive application architecture to handle variability in the solution space.

Acknowledgements

This work has been partially supported by FINEP/MCTI under grant no. 03.14.0062.00.

8. REFERENCES

[1] P. Mell and T. Grance, "The NIST Definition of Cloud Computing," NIST, Tech. Rep., 2011.

[2] B. Rimal, A. Jukan, D. Katsaros, and Y. Goeleven, "Architectural Requirements for Cloud Computing Systems: An Enterprise Cloud Approach," *Journal of Grid Computing*, vol. 9, no. 1, pp. 3–26, 2011.

[3] F. J. van der Linden, K. Schmid, and E. Rommes, *Software product lines in action.* Springer, 2007.

[4] P. C. Clements and L. M. Northrop, *Software Product Lines: Practices and Patterns.* Addison-Wesley Longman Publishing Co., Inc., 2003.

[5] K. Schmid and A. Rummler, "Cloud-based software product lines," in *Proceedings of the International*

Software Product Line Conference, vol.2, 2012, pp. 164–170.

[6] K. C. Kang, S. Kim, J. Lee, K. Kim, E. Shin, and M. Huh, "FORM: A feature-oriented reuse method with domain-specific reference architectures," *Annals of Software Engineering*, vol. 5, no. 1, pp. 143–168, 1998.

[7] J. van Gurp, J. Bosch, and M. Svahnberg, "On the Notion of Variability in Software Product Lines," in *Proceedings of the Working Conference on Software Architecture.* IEEE Computer Society, 2001, p. 45.

[8] K. Pohl, G. Böckle, and F. Van Der Linden, *Software product line engineering.* Springer, 2005, vol. 10.

[9] K. Czarnecki, S. Helsen, and U. Eisenecker, "Staged Configuration Using Feature Models," in *Software Product Lines*, ser. Lecture Notes in Computer Science, R. Nord, Ed. Springer Berlin Heidelberg, 2004, vol. 3154, pp. 266–283.

[10] "SEI software product line framework," available from: URL: http://www.sei.cmu.edu/productlines/.

[11] H. Gomaa, *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures.* Addison-Wesley, 2004.

[12] N. Medvidovic and R. N. Taylor, "A classification and comparison framework for software architecture description languages," *Software Engineering, IEEE Transactions on*, vol. 26, no. 1, pp. 70–93, 2000.

[13] M. Marschall, *Chef Infrastructure Automation Cookbook.* Packt Publishing Ltd, 2013.

[14] L. P. Tizzei, M. Dias, C. M. F. Rubira, A. Garcia, and J. Lee, "Components meet aspects: Assessing design stability of a software product line," *Information and Software Technology*, vol. 53, no. 2, pp. 121 – 136, 2011.

[15] C. Gacek and M. Anastasopoules, "Implementing product line variabilities," in *Proceedings of the Symposium on Software Reusability: Putting Software Reuse in Context.* NY, USA: ACM, 2001, pp. 109–117.

[16] G. Reese, *Cloud application architectures: building applications and infrastructure in the cloud.* O'Reilly Media, Inc., 2009.

[17] J. Schroeter, P. Mucha, M. Muth, K. Jugel, and M. Lochau, "Dynamic configuration management of cloud-based applications," in *Proceedings of the International Software Product Line Conference - Vol.2.* ACM, 2012, pp. 171–178.

[18] M. Fantinato, M. de Toledo, and I. Gimenes, "A feature-based approach to electronic contracts," in *Proceedings of the International Conference on E-Commerce Technology and The International Conference on Enterprise Computing, E-Commerce, and E-Services*, 2006, pp. 34–34.

[19] I. Sommerville, *Software Engineering (7th Edition).* Pearson Addison Wesley, 2004.

[20] K. Czarnecki, S. Helsen, and U. Eisenecker, "Staged configuration through specialization and multilevel configuration of feature models," *Software Process: Improvement and Practice*, vol. 10, no. 2, pp. 143–169, 2005.

[21] T. Bednasch, "Konzept und implementierung eines konfigurierbaren metamodells für die merkmalmodellierung. Diplomarbeit, Fachbereich Informatik, Fachhochschule Kaiserslautern,Standort Zweibrücken, Germany," 2002.

[22] D. Benavides, P. Trinidad, and A. Ruiz-Cortés, "Automated reasoning on feature models," in *Advanced Information Systems Engineering.* Springer, 2005, pp. 491–503.

[23] S. Walraven, D. V. 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*, 2014.

[24] S. T. Ruehl and U. Andelfinger, "Applying software product lines to create customizable software-as-a-service applications," in *Proceedings of the International Software Product Line Conference, vol. 2.* ACM, 2011, p. 16.

[25] J. Segal and C. Morris, *Handbook of Research on Computational Science and Engineering: Theory and Practice.* USA: IGI Global, 2011, vol. 1, ch. Developing Software for a Scientific Community:Some Challenges and Solutions, pp. 177–196.

[26] J. Lee and G. Kotonya, "Combining service-orientation with product line engineering," *Software, IEEE*, vol. 27, no. 3, pp. 35–41, 2010.

[27] P. Laplante, J. Zhang, and J. Voas, "What's in a name? distinguishing between SaaS and SOA," *IT Professional*, vol. 10, no. 3, pp. 46–50, 2008.

[28] I. Kumara, J. Han, A. Colman, T. Nguyen, and M. Kapuruge, "Sharing with a difference: Realizing service-based SaaS applications with runtime sharing and variation in dynamic software product lines," in *International Conference on Services Computing.* IEEE, 2013, pp. 567–574.

[29] O. Haugen, B. Møller-Pedersen, J. Oldevik, G. K. Olsen, and A. Svendsen, "Adding standardized variability to domain specific languages," in *Proceedings of the International Software Product Line Conference*, 2008, pp. 139–148.

[30] A. Nascimento, C. Rubira, and F. Castor, "Using CVL to support self-adaptation of fault-tolerant service compositions," in *Proceedings of the International Conference on Self-Adaptive and Self-Organizing Systems*, 2013, pp. 261–262.

[31] R. Mietzner, "A method and implementation to define and provision variable composite applications, and its usage in cloud computing," Ph.D. dissertation, Stuttgart University, 2010.

[32] E. Wittern, J. Kuhlenkamp, and M. Menzel, "Cloud service selection based on variability modeling," in *Service-Oriented Computing.* Springer, 2012, pp. 127–141.

[33] C. Quinton, N. Haderer, R. Rouvoy, and L. Duchien, "Towards multi-cloud configurations using feature models and ontologies," in *Proceedings of the International workshop on Multi-cloud applications and federated clouds.* ACM, 2013, pp. 21–26.

[34] J. Schroeter, S. Cech, S. Götz, C. Wilke, and U. Aßmann, "Towards modeling a variable architecture for multi-tenant saas-applications," in *Proceedings of the international workshop on variability modeling of software-intensive systems.* ACM, 2012, pp. 111–120.