International Journal of Cloud Applications and Computing
Volume 9 • Issue 4 • October-December 2019

Policy-Driven Middleware for Multi-Tenant SaaS Services Configuration

Khadija Aouzal, SEEDS Team, STRS Lab, INPT, Rabat, Morocco
Hatim Hafiddi, SEEDS Team, STRS Lab, INPT, Rabat, Morocco & IMS Team, ADMIR Lab, ENSIAS, Rabat, Morocco
Mohamed Dahchour, SEEDS Team, STRS Lab, INPT, Rabat, Morocco

ABSTRACT

The multi-tenancy architecture allows software-as-a-service applications to serve multiple tenants with a single instance. This is beneficial as it leverages economies of scale. However, it does not cope with the specificities of each tenant and their variability; notably, the variability induced in the required quality levels that differ from a tenant to another. Hence, sharing one single instance hampers the fulfillment of these quality levels for all the tenants and leads to service level agreement violations. In this context, this article proposes a policy-driven middleware that configures the service according to the non-functional requirements of the tenants. The adopted approach combines software product lines engineering and model driven engineering principles. It spans the quality attributes lifecycle, from documenting them to annotating the service components with them as policies, and it enables dynamic configuration according to service level agreements terms of the tenants.

KEYWORDS

MDE, Multi-Tenancy, Non-Functional Variability, Policy, SaaS, SLA, SPLE

INTRODUCTION

Software as a Service (SaaS) applications are generally built based on the multi-tenancy architecture (Kabbedijk et al., 2015). This architecture style allows serving multiple tenants with a single instance of the application. Hence, it enables, among others, high utilization of resources, easy maintenance and cost efficiency. However, it limits the support of service variability among tenants, since the same instance of the SaaS service is shared between them and the particularities of each tenant's needs are not considered. To cope with this, customization and adaptation techniques must be enforced to get SaaS services tailored to tenants' needs and their specific contexts, while maintaining multi-tenancy.

Note that not only has the SaaS service to be adapted to functional requirements of the different tenants, but it has also to meet non-functional requirements in their various levels. In fact, the required service quality level may differ from a tenant to another. For example, a tenant can just require minimal security level as it needs solely to be authenticated and has no other security concerns; while another may require a high security level and advanced and supplementary security mechanisms such as access control and encryption. There are also situations when the same tenant changes its desired quality level through time; for instance, when there is an increase of service requests in a critical period, the tenant may want to decrease the response time and to increase the uptime value in order to smoothly perform his requests. Therefore, the application needs to be adapted dynamically in order to meet every tenant's quality requirements as if he is the only one to consume the service.

However, configuring the service utterly for each tenant, at a time, is a heavy and tedious task for developers and architects. Thus, commonalities and variations among tenants regarding quality

DOI: 10.4018/IJCAC.2019100105

attributes should be first captured. This is one of the principles of Software Product Lines Engineering (SPLE) that enables high reusability in shorter time, at lower costs and with higher quality (Pohl et al., 2005). The SPLE paradigm allows creating product families or lines sharing commonalities and differing in some aspects. These aspects are generally represented as variation points, in most of the variability modeling languages, e.g. Feature Model (FM) (Batory, 2005), Orthogonal Variability Modeling (OVM) (Pohl et al., 2005), Extended Feature Model (Benavides et al., 2005), etc. In order to derive the final product tailored to specific needs, configuration and adaptation mechanisms are performed at the level of variation points while considering the dependencies between all the different artifacts composing the final product and the product line as well. SPLE involves two complementary phases: Domain Engineering and Application Engineering. Domain Engineering allows the management of a product line or family as a whole and not as individual products. It follows "for reuse" development strategy, starting by domain analysis to identify the commonalities and variabilities among the requirements, and aiming at defining the architecture of a product line, its reusable artifacts and the relationships and dependencies between them. Application Engineering, on the other hand, follows "by reuse" development strategy. In this phase the final products are derived by configuring the artifacts defined in the Domain Engineering phase to meet specific requirements. It allows also conformity checking to validate that the final derived product is conformed to the specific needs it intends to respond to.

The proposed approach uses the SPLE principles to build SaaS services tailored to tenant-specific Service Level Agreements (SLAs). In a previous work (Aouzal et al., 2018), the authors defined the Domain Engineering phase to build SLA families. For that purpose, they combined Model Driven Engineering (MDE) and SPLE to model Non-Functional Requirements as features composing the FM. Based on domain analysis, the NFRs of the tenants, their commonalities, variation points and variants are modeled according to the proposed NFVariability Metamodel. Model-to-model transformations are then performed to generate the corresponding Generic SLA in conformance with the proposed VariableSLA metamodel. The Generic SLA is a document that contains the terms and the Service Level Objectives (SLOs) of all the contracting tenants, i.e. it encompasses mandatory terms and all the variants for the optional terms.

In this paper, the authors will continue on this SPLE-MDE synergy while using EFM to model the non-functional properties attributed with their corresponding tenants. They propose a middleware responsible for intercepting tenants' requests and mediating them to the corresponding components of the application, configured with adequate quality attributes. To achieve that, the middleware generates core and tenant-specific policies from the Generic SLA. These policies constitute the reference for the Configurator component to adapt the application according to the defined policies. Those configurations are based on annotations in the form of key-value pairs. This annotation-based approach is used for its simplicity and flexibility (Nosál & Porubän, 2014). As contributions of this paper: the authors present the architecture of the proposed middleware and its different components; they define a Policy metamodel, in order to model policies retrieved from the Generic SLA as annotations, along with model-to-model transformations from VariableSLA metamodel to Policy metamodel; and they propose a Configurator metamodel that models and describes how the Configurator component of the middleware configures the SaaS service. The proposition in its all phases is illustrated by a case-study.

The rest of the paper is structured as follows. Section 2 presents the motivating scenario that illustrates the problem. An overview of the proposed approach is described in Section 3, and the middleware architecture is presented in Section 4. Section 5 presents the model-to-model transformations performed to generate tenant-specific policies. The policy-driven service configuration is depicted in Section 6. Section 7 discusses and evaluates the approach. Section 8 presents related work. Finally, Section 9 concludes the paper and presents future work.

MOTIVATION AND CHALLENGES

This paper aims at defining a middleware for runtime adaptation of SaaS services according to quality attributes of the on-boarding tenants. The middleware should be able to derive policy-driven service instances for tenants having different quality levels, to monitor them and ensure runtime adaptation. This section presents a motivating scenario that illustrates non-functional variability and that brings out the requirements and challenges to be taken into consideration by the middleware.

Scenario

The scenario consists in a document management application that enables its tenants and end users to create, manage and store their documents: contracts, financial documents, reports, etc. It provides them tools for controlling and having visibility on modifications made to documents, on their status during the process of review, negotiation, approval or signature. All the activities made to a document are kept synchronized between the parties involved in those activities for that document to ensure that data are up to date and accurate.

The application is implemented following the SaaS model and the multi-tenancy architecture. This architecture introduces several challenges regarding the quality of service and tenant isolation. In fact, sharing the same instance between multiple tenants may affect service performance and dependability and may threat tenants' data security. Moreover, its one-size-fits-all approach does not cope with the non-functional variability present in the application. Non-functional variability can be classified in two categories: coarse-grained variability and fine-grained variability. Coarse-grained variability consists in variability in the different concerns that a quality attribute is composed of, e.g. security has as concerns authentication, access control, encryption, etc. Fine-grained variability, on the other hand, refers to variations in the required quality levels that differ from a tenant to another, e.g. an uptime of 99% for tenant1 and 95% for tenant2.

For the document management application, it should at least ensure, the following quality attributes. In terms of security, the concerns are: 1) authentication: the user must be authenticated to access the application using either username and password credentials or QR code to use the application in other devices; 2) data integrity and privacy: data of a tenant must not be altered or modified by non-authorized users, and must be kept private from other tenants; 3) authorization and access control: authorization and access rules must be defined to restrict the access to certain components and features of the application to user roles that are allowed to; 4) session duration and time management: the access might be restricted in duration according to the access rules assigned to user roles. In addition to security, the application must fulfill certain levels of other non-functional properties such as response time, availability and service adaptability according to the used device.

The application has to be tailored to the variant levels of non-functional properties that characterize each tenant. For instance, consider three tenants: tenant 1, tenant 2 and tenant 3. Table 1 shows the quality levels required by each tenant.

Challenges

Non-functional variability in SaaS services introduces several challenges that should be addressed starting from non-functional requirements modeling to policy-driven service derivation. This section describes those challenges and requirements.

Challenge 1 - Modeling Non-functional requirements and their variability: The approach should enable the fact that the non-functional requirements of the application are captured and modeled as features in terms of SPLE. This modeling allows the separation of commonalities and variabilities, and determining fine-grained variable assets and coarse-grained ones. The modeling should take into account the dependencies between the different quality attributes to help managing conflicts between them at early stages.

Table 1. Quality levels required by the different tenants

Tenant	Security				Performance		Adaptability Device Awareness
	Authentication	Data Integrity and Privacy	Access Control	Session Duration and Time Management	Availability	Response Time	
Tenant 1	+	+	+	-	99.95%	500 ms	+
Tenant 2	+	+	-	-	99%	300 ms	-
Tenant 3	+	+	+	+	98%	800 ms	-

Challenge 2 - Deriving tenant-specific Policies: As SLA is the document that describes the service quality levels agreed upon with the tenants, the transition to the elaboration of tenant-specific policies should be easy and smooth, after modeling the non-functional requirements. Thus, the approach should allow automatic generation of policies after domain analysis and documenting the non-functional requirements of the different tenants. The generated policies must be tenant-specific in order to preserve the specificities of each tenant regarding the quality attributes.

Challenge 3 - Configuring the SaaS service dynamically according to quality attributes at runtime: The approach should provide tools for ensuring dynamic configuration of the SaaS service according to the defined policies. For that, the application should be built in a way that configurations are per-tenant and isolated. Actually, a configuration for a tenant should not affect the performance of the application for the other tenants and interfere with other configurations. Moreover, the adaptation should be maintained all over the service lifecycle and during time evolution to involve eventual context and requirements changes. Thus, the application must be monitored to detect environment changes and SLA violations.

In order to address those challenges, the authors propose a model-driven approach combined with the SPLE paradigm, as it will be presented in the subsequent sections. Section 5 will particularly depict how the proposed approach deals with each one of these challenges, and to what extent it handles each one of them.

APPROACH OVERVIEW

As mentioned above, the approach combines SPLE and MDE paradigms to manage non-functional variability of SaaS services. It involves the two SPLE phases: domain engineering and application engineering, and is based on a set of metamodels and model-to-model transformations. Figure 1 depicts an overview of the proposed approach.

Domain Engineering

Domain Engineering process aims at defining the commonalities and variabilities regarding an application or a service. In the context of non-functional variability in SaaS services, this process begins with a domain analysis to retrieve non-functional requirements of each tenant. These non-functional requirements are then dealt with as features and modeled using the EFM. This latter is a tree-structured diagram that models quality attributes as mandatory or optional features along with possible attributes specifying extra-properties of these features, and that specifies the constraints between the features. In the proposed approach, the EFM is in conformance with NFVariability metamodel defined in a previous work (Aouzal et al., 2018). This metamodel allows modeling QoS characteristics as variable features that can be either mandatory or optional, and the inter- and intra-dependencies between them. Figure 2 shows the instance of this metamodel, the EFM, for the document management application. In this EFM, both coarse-grained and fine-grained variability are identified. Obviously, the EFM is constructed for each service the application is composed of. Due to simplicity reasons, the document management application is considered as one service.

Figure 1. Approach overview

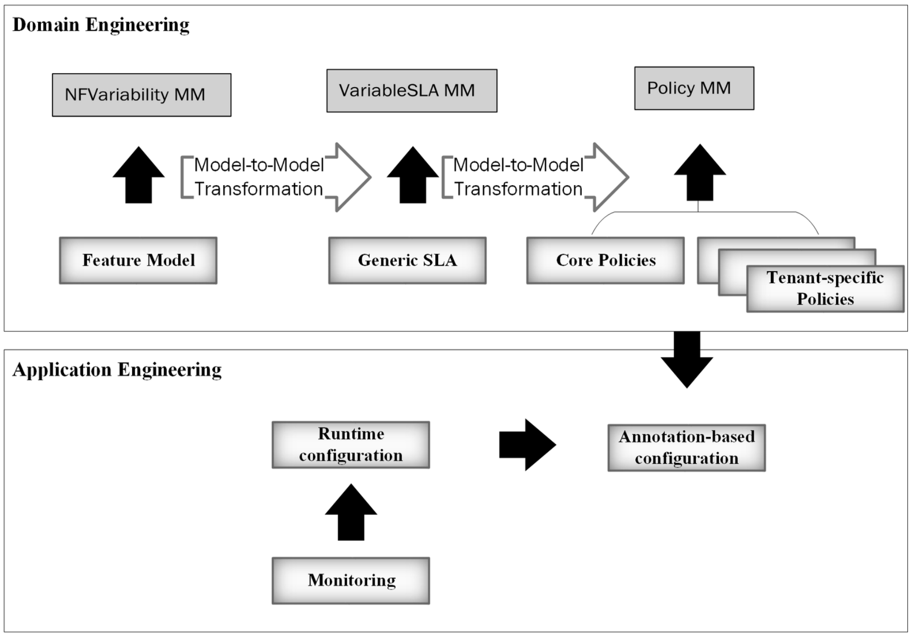
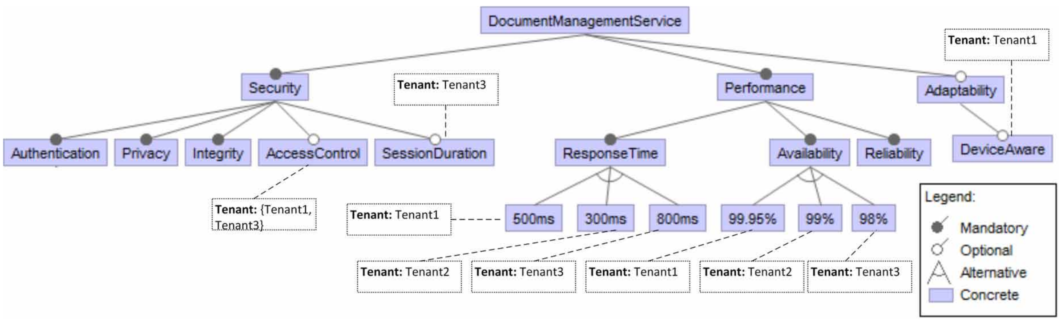


Figure 2. Extended feature model for non-functional variability in document management service



The aim of the approach is to build SLA families, i.e. SLAs that share commonalities and differ in some respects. For this purpose, VariableSLA metamodel is proposed to define the terms of an SLA as variables. The instance of this metamodel is the Generic SLA that contains the terms and the SLOs of all the tenants that consume the service. It is generated from the EFM through model-to-model transformations. These transformations rules are described in detail in (Aouzal et al., 2018) along with the involved models and metamodels.

The Generic SLA represents the base to construct core and tenant-specific policies. It is by means of model-to-model transformations that those policies are generated, by defining VariableSLA as the source metamodel and Policy metamodel as the target one. Those metamodels along with the transformations are more elaborated in Section 5.

Application Engineering

Application Engineering process consists in building configurable services by reusing and exploiting the assets of the Domain Engineering process. For the current context, these reusable assets refer to the core and the tenant-specific policies. They are the base of the dynamic configuration so as to inform the middleware, responsible for configuring and adapting the service, about the required policies in the form of annotations. The configuration can be classified in two categories: static configuration and dynamic configuration. The static configuration refers to configurations made by the developer at design time, manually or semi-manually, based on the defined policies. Dynamic configuration, on the other hand, concerns runtime adaptation that is made dynamically by injecting tenant-specific policies into the adequate components the service is composed of and calling on defined actions to make the configurations that match the policies. Moreover, runtime adaptation may be triggered by context changes. Those changes are raised by means of application monitoring which checks the conformity of the running service for a specific tenant to its corresponding policies.

MIDDLEWARE ARCHITECTURE

The middleware is responsible for configuring the SaaS service according to non-functional requirements of tenants. As shown in Figure 3, it has as input the Generic SLA and it is composed of: Policies Generator, Policies Repository, Configurator, Rules Engine and the Monitoring System. The Policies Generator generates policies from the Generic SLA. Those policies are stored in the Policies Repository. The Configurator is composed of the Annotator, the Reconfigurator and the Action Decider. The Annotator annotates the corresponding components of the SaaS service with adequate policies. The Action Decider decides of which action to take in response to the alerts received from the Monitoring System. These actions are sent to the Reconfigurator that reconfigures the components according to them. The Monitoring System monitors the service and its execution environment, and checks the conformance of that service to the policies defined in the Policies Repository. The Rules Engine contains pre-defined rules, subject to modification over time, which define which actions to be performed in response to SLA violation events.

The configuration process concerns two phases: design time and runtime. At design time, the Policies Generator generates policies from the Generic SLA. The SLA is transformed to Policies in order to retrieve relevant information about non-functional properties expressed in SLA, i.e. their names and values, to categorize them according to how they are applied at the level of the application. These policies are stored in the Policies Repository. The Configurator gets the core policies from this repository and annotates the components of the SaaS service, through the Annotator component, with those policies and makes corresponding configurations, see Figure 4.

At runtime, the Configurator annotates the SaaS application components with the specificities of the tenant sending the request, i.e. its specific policies. Actually, see Figure 5, when a tenant wants to consume the service, the Configurator gets the policies that concern that tenant and service from the Policies Repository and annotates, at runtime, the components with those tenant-specific policies. While the service is running, the monitoring system keeps checking the conformity of the service to the agreed SLA and sends alerts to the Configurator in case of non-conformity. In that case, the Configurator via the Action Decider gets rules that match the alerts from the Rules Engine. According to these rules, the Action Decider decides of the adequate actions to be taken by the Reconfigurator in order to reconfigure dynamically the service to meet the required quality levels.

In the next section, the Policies Generator and the Configurator components are elaborated and illustrated with the case study introduced previously.

CORE AND TENANT-SPECIFIC POLICIES GENERATION

The policies generation relies on The Policies Generator component of the middleware. This component enables Generic SLA transformation to Policies. By this, the authors adhere to the principle of Concerns Separation: as an output of the generator, the policies document plays the role of a reference that informs the configurator about the tenant-specific SLA state. Thus, the transformation of Generic SLA to policies is made following two steps: globally to generate Core Policies which is

Figure 3. Middleware architecture

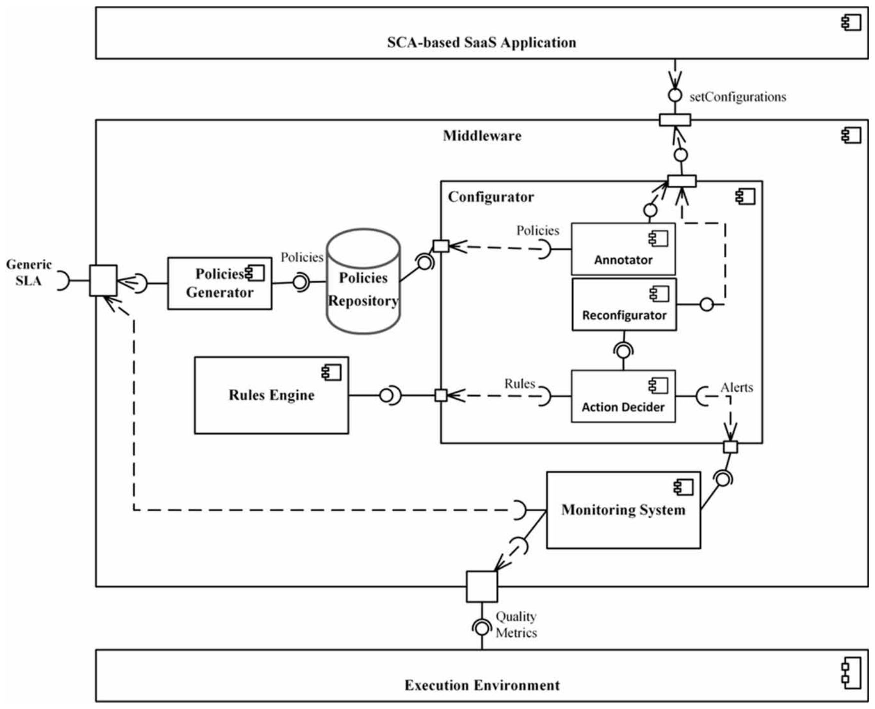
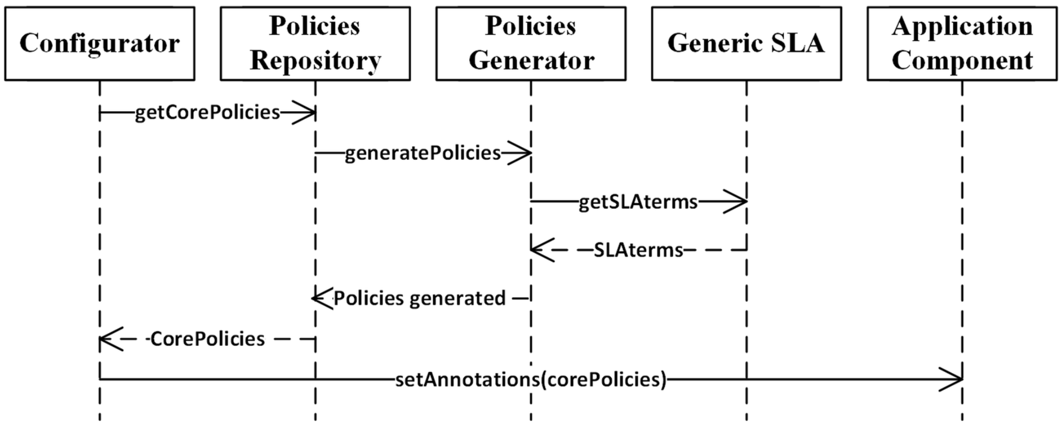


Figure 4. Design time configuration sequence diagram



done once for all the tenants and per-tenant basis to retrieve tenant-specific policies of each tenant. In the following, the VariableSLA and Policy metamodels are defined and then the SLA2Policy transformations are performed.

VariableSLA Metamodel

The VariableSLA metamodel, see Figure 6, is structured in two main blocks: Contracting parties and Cloud services. Terms are related to each cloud service; they represent what was agreed upon during SLA negotiation. They can be Core terms or Specific Terms. Core Terms specify the terms that are common to all tenants involved in the contract. Specific Terms concern what is specific to each tenant. Each term corresponds to a quality attribute and is constituted of one or more SLOs that specify the quality level expected by the service. The SLO is constrained by one or more preconditions, which has to be satisfied to ensure the requested quality level.

Policy Metamodel

This metamodel is depicted in Figure 7. The meta-class *Policy* represents the root of this metamodel. The policies are represented as annotations. An Annotation can be either *SimpleAnnotation* or *AttributedAnnotation*. *SimpleAnnotation* defines the annotations that do not have attributes and that annotate other annotations. The *AttributedAnnotation* metaclass represents attributed annotations to refer to non-functional properties and their values. An *AttributedAnnotation* can be of *interaction*, *implementation* or *elasticity* type. An annotation of interaction type means that it affects the communication and the message exchange between two components, i.e. the binding wire. For example, an encryption annotation requires that the communication between the service component and the reference component must be encrypted. Hence, it is an interaction annotation. An annotation of implementation type denotes that it affects the behavior of the container that implements the component and the code used to run it. Running a component in a transaction is an example of such annotation. As for the annotation of elasticity type, it concerns the non-functional properties that deal with the service scalability, e.g. in order to ensure a short response time the service needs to be scaled up by adding more resources or scaled out by adding more virtual machines. The constraints between *AttributedAnnotation* instances are expressed by the relationships: requires and excludes. Components are tied to attributed annotations through the *appliesTo* reference.

Annotations availability time depends on the policies they will represent and their optionality (core or specific). Core policies are annotated to the corresponding target components during

Figure 5. Runtime configuration sequence diagram

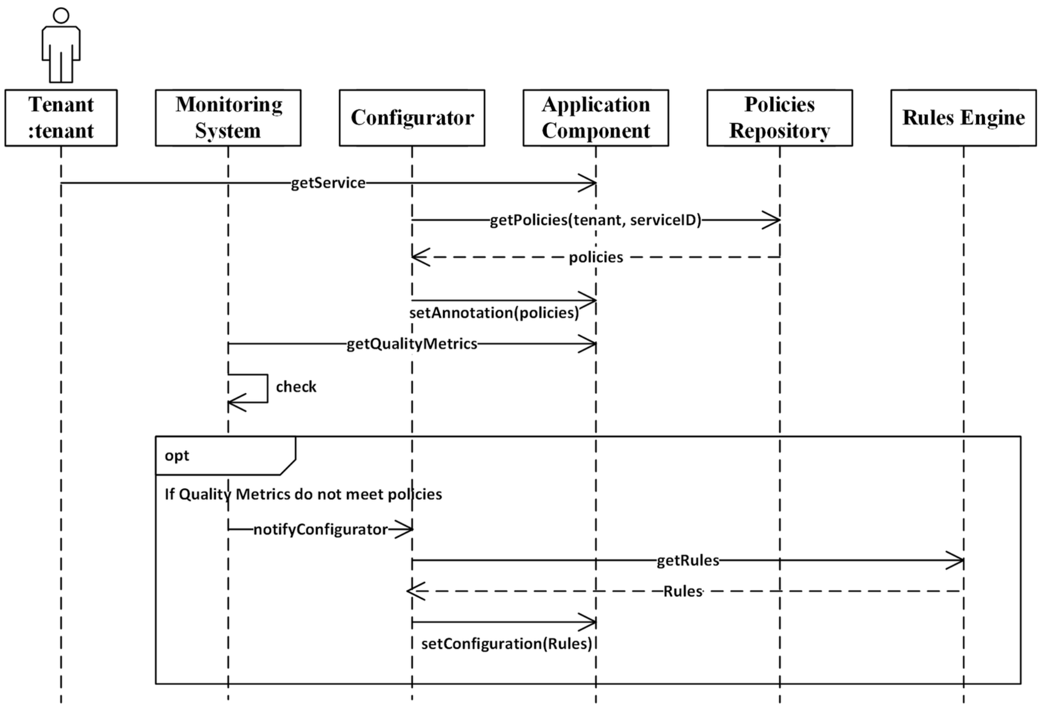
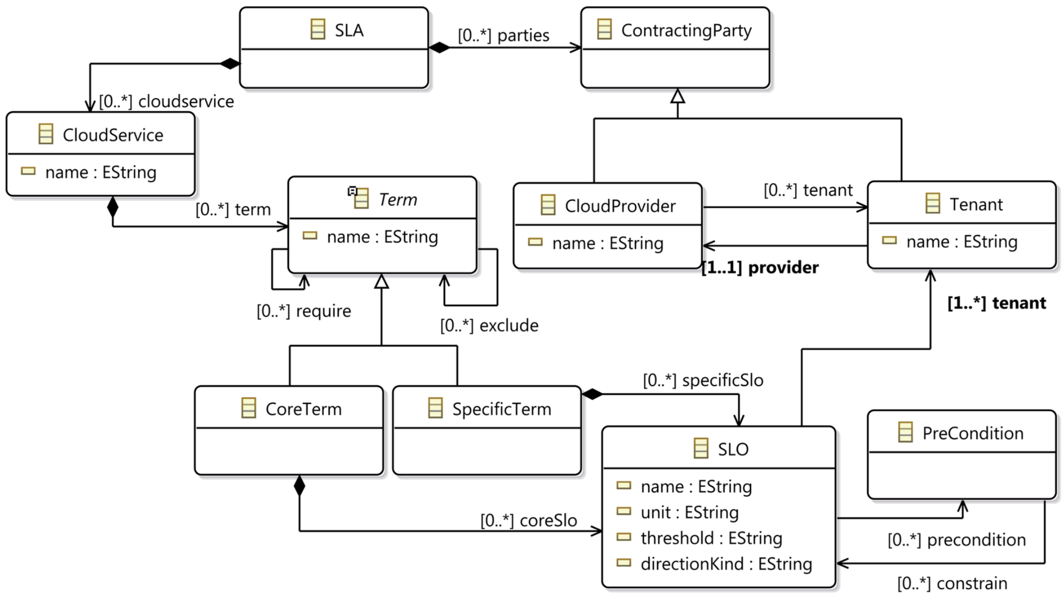


Figure 6. VariableSLA metamodel



implementation time. On the other hand, specific policies, that concern all the tenants but differ in values from a tenant to another, are defined in a generic way and they are used and made available at runtime instantiated with values that specify the tenant the request is related to. In Java, for example, this is possible with @Retention annotation that determines when the defined annotation should be available (Guerra et al., 2010). The values, which are attributed to these annotations at runtime, are retrieved by the Configurator from the Policies Repository by selecting the ones matching the required service and the tenant that triggers the request.

Model-to-Model Transformation Specification

The model-to-model transformations are performed from SLA as input model to policies as output model, see Figure 8. The transformation rules are described in the following.

The SLA metaclass, from VariableSLA metamodel, is mapped to Policy metaclass of Policy metamodel. The subsequent elements of Policy are retrieved after defining and executing the transformation rules from their corresponding input elements. CoreTerm and SpecificTerm are transformed to SimpleAnnotation to construct annotations named respectively core and specific. These simple annotations will annotate attributed ones by creating adequate AttributedAnnotation instances along with the corresponding properties and references: name, excludes, requires, etc. Listing 1 defines these rules using Epsilon Transformation Language (ETL) (Kolovos et al., 2018).

Attribute instances of target model are retrieved from their equivalent SLO instances along with mapping the name property of a target element to the name property of its source counterpart, and the value property of an Attribute instance to the threshold property concatenated with the unit property, if defined, of an SLO instance. Listing 2 shows how this transformation is defined using ETL.

Listing 1. CoreTerm and SpecificTerm to SimpleAnnotation transformation rule using ETL

```
rule CoreTerm2SimpleAnnotation
transform s: variableSLA!CoreTerm
to t: policy!SimpleAnnotation {
```

Figure 7. Policy metamodel

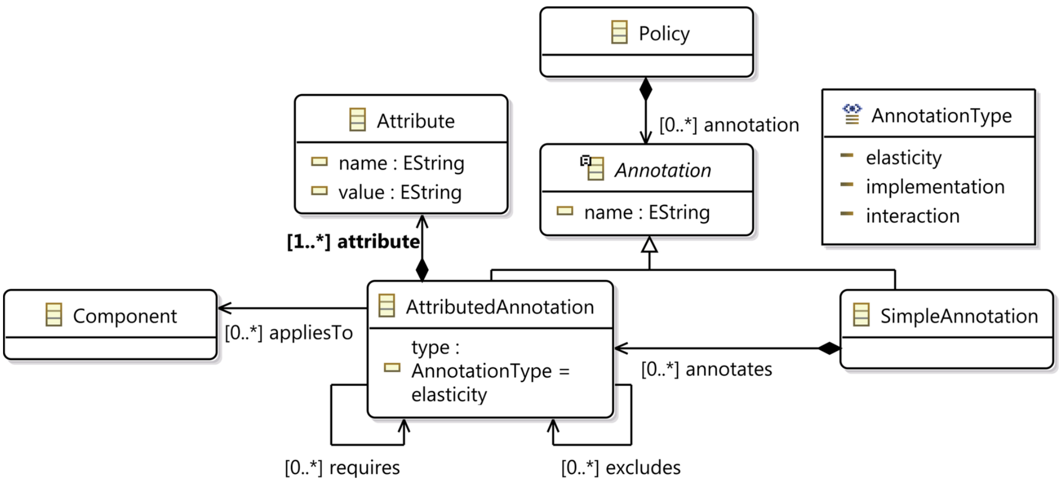
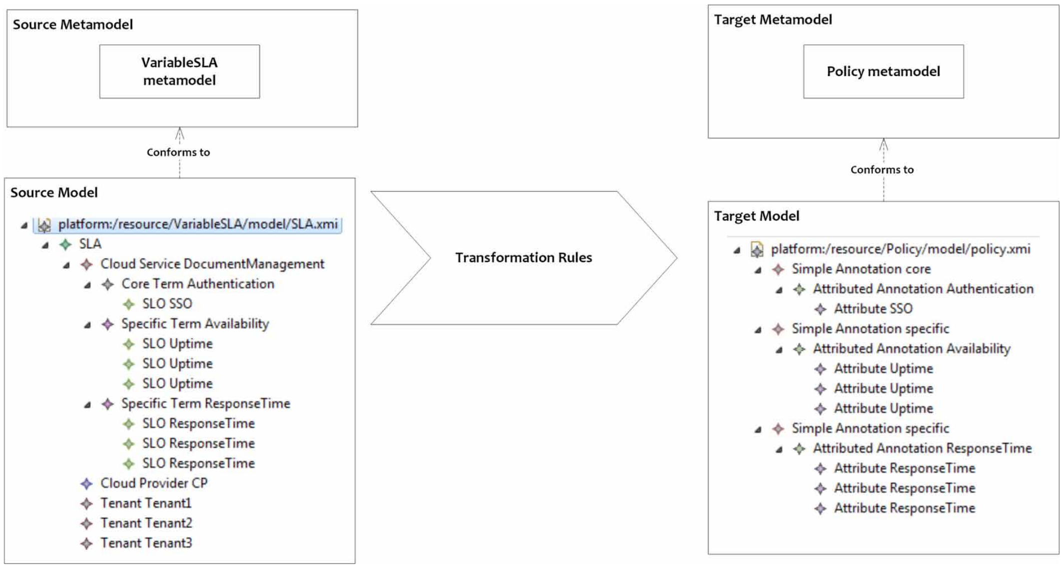


Figure 8. Model-to-model transformation mechanism



```
platform:/resource/VariableSLA/model/SLA.xmi
  SLA
    Cloud Service DocumentManagement
      Core Term Authentication
        SLO SSO
      Specific Term Availability
        SLO Uptime
        SLO Uptime
        SLO Uptime
      Specific Term ResponseTime
        SLO ResponseTime
        SLO ResponseTime
        SLO ResponseTime
    Cloud Provider CP
    Tenant Tenant1
    Tenant Tenant2
    Tenant Tenant3
```

```
platform:/resource/Policy/model/policy.xmi
  Simple Annotation core
    Attributed Annotation Authentication
      Attribute SSO
  Simple Annotation specific
    Attributed Annotation Availability
      Attribute Uptime
      Attribute Uptime
      Attribute Uptime
  Simple Annotation specific
    Attributed Annotation ResponseTime
      Attribute ResponseTime
      Attribute ResponseTime
      Attribute ResponseTime
```

```
        t.name="core";
        var annot = new policy!AttributedAnnotation;
        annot.name = s.name;
        annot.excludes = s.exclude;
        annot.requires = s.require;
        annot.attribute = s.coreSlo.equivalent();
        t.annotates.add(annot);
}
rule SpecificTerm2SimpleAnnotation
transform s: variableSLA!SpecificTerm
```

```
to t: policy!SimpleAnnotation{
        t.name="specific";
        var annot = new policy!AttributedAnnotation;
        annot.name = s.name;
        annot.excludes = s.exclude;
        annot.requires = s.require;
        annot.attribute = s.specificSlo.equivalent();
        t.annotates.add(annot);
}
```

Listing 2. SLO2attribute transformation rule using ETL

```
rule SLO2Attribute
transform s:variableSLA!SLO
to t:policy!Attribute{
        t.name = s.name;
        if (s.unit.isDefined()){
        t.value = s.threshold + s.unit;
        }
        else t.value = s.threshold;
}
```

Illustrative Example

Figure 9 represents a sample of the Generic SLA related to the document management application. For simplicity reasons, just some quality attributes are taken into consideration: Authentication, Availability and Response Time. The authentication attribute is mandatory, i.e. an instance of CoreTerm. It has to be present in the corresponding components for all the tenants. The authentication mechanism used by all the tenants is Single Sign-On (SSO); this is modeled as a SLO. Availability and Response Time are, on the contrary, specific terms whose corresponding SLOs, uptime and response time respectively, are tenant-specific and differ from one to another. This model constitutes the input of the transformation engine. The structure of the output model is depicted in Listing 3. In this model, it is shown that Authentication being a core term is annotated with "core", and the two other specific terms (Availability and Response Time) are annotated with "specific". The illustrated quality attributes are modeled as attributed annotations as they have attributes.

Listing 3. Example of policy document for document management service

```
<policy:SimpleAnnotation name="core">
    <annotates name="Authentication">
      <attribute name="SSO"/>
    </annotates>
  </policy:SimpleAnnotation>
  <policy:SimpleAnnotation name="specific">
    <annotates name="Availability">
      <attribute name="Uptime" value="99.95%" tenant=" tenant1"/>
      <attribute name="Uptime" value="99%" tenant=" tenant2"/>
      <attribute name="Uptime" value="98%" tenant=" tenant 3"/>
    </annotates>
  </policy:SimpleAnnotation>
  <policy:SimpleAnnotation name=”specific”>
    <annotates name=”ResponseTime”>
      <attribute name=”ResponseTime” value=”500ms” tenant=” tenant
1”/>
      <attribute name=”ResponseTime” value=”300ms” tenant=”
tenant2”/>
      <attribute name=”ResponseTime” value=”800ms” tenant=” tenant
3”/>
    </annotates>
  </policy:SimpleAnnotation>
```

Using annotation-based notation, the policies will become such as the following.

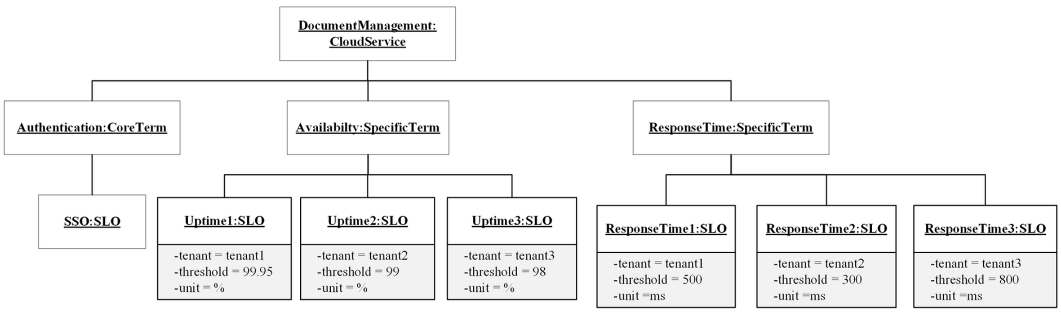
```
@core
@Authentication = “SSO”
@specific
@Availability(Uptime = “99.95%”)
@specific
@ResponseTime(Response time = “500ms”)
```

It is worth noting that the annotations’ attributes for the specific policies are instantiated dynamically with their values by the Configurator at runtime on a per-tenant basis.

POLICY-DRIVEN SERVICE CONFIGURATION

Basically, the SaaS application is based on Service Component Architecture (SCA). This architectural style is advocated due to its benefits in terms of reuse, composition and dynamic configuration ensured thanks to runtime binding and unbinding of components. It consists of a set of specifications that describe how applications can be built in conformance to the principles of Service Oriented Architecture (SOA) and in an independent platform way. The building blocks of such architectures are components that may require certain functions of other components to provide their own capabilities. The interfaces responsible for providing and retrieving functions are services and references, respectively (OASIS, 2011). The SCA development process encompasses two phases: Implementation and Assembly. The former phase consists in implementing components which offer services and consume others. The latter allows assembling components through connecting references to services in order to build business applications. SCA allows service portability across different infrastructures by decoupling service implementation and assembly from the details of infrastructure capabilities (OASIS, 2011). SCA can also be used in the context of SaaS applications, since they are generally built following SOA architecture.

Figure 9. A sample of the Generic SLA related to the document management application



The configurator component aims at defining configurations and actions to be taken in order to meet the quality level defined in the SLA. It takes the policies generated by the Policies Generator from the Policies Repository, annotates the components of the SCA-based SaaS application with them and performs global and per-tenant adaptations according to the core policies and the tenant-specific policies, respectively.

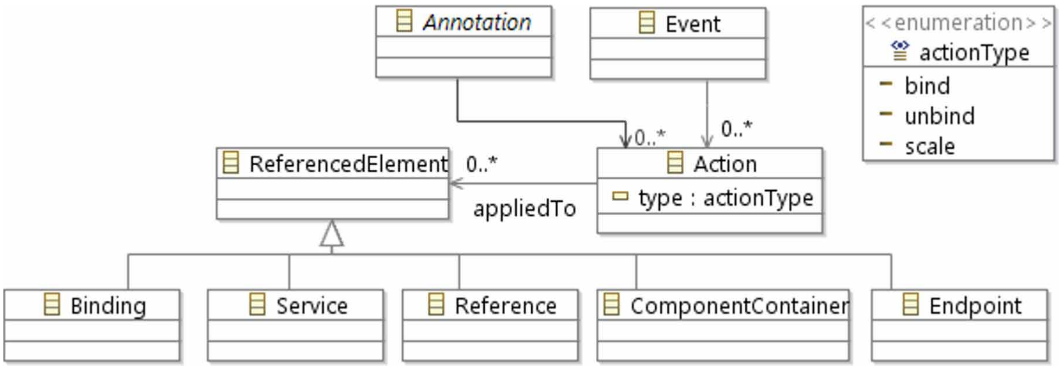
Configurator Metamodel

The corresponding metamodel is described in Figure 10. The configurator has two sources that may trigger actions: Annotation and Event. Annotations represent the set of policies that should be set to the components. Based on these annotations, the configurator takes actions on multiple elements so as to respect the corresponding policies. The events represented by Event concern the alerts notified to the configurator by the Monitoring System. The performed actions, which depend on the alerts and the rules defined in the Rules Engine, can be any of these types: bind, unbind or scale; and they are applied to different elements generalized by the meta-class ReferencedElement. These elements can be: a Binding, i.e. the communication link between a service and a reference; a Service offered by a component; a Reference consumed by a component; a ComponentContainer which runs and executes the code; or an Endpoint which represents the communication points of a binding. The actions relate also to the type of the policy: elasticity, implementation and interaction. For instance, a policy of elasticity type will certainly lead to a scale action, but it may also involve other actions. The action types are not limited to: bind, unbind and scale; they are extensible to encompass other possible actions that can rely on e.g. the Cloud Offering and its resource management system, especially for non-functional requirements that are directly related to cloud resource management such as availability and performance. The related actions can be supported by, for example, load balancing, fault tolerance based resource allocation, workload management, elasticity, etc (Jennings & Stadler, 2014).

Illustrative Example

Concerning the document management application and its aforementioned related policies, the components in question that the application is composed of are annotated with those policies mentioning the optionality of the quality attribute (core or specific), the quality level and its scope of action (elasticity, interaction or implementation). For the authentication quality attribute, it is a core policy and of interaction type; that is, the configurator will bind the component in question to an existing component implementing the authentication mechanism with the SSO method. This is done globally for all the tenants, i.e. they all have access to the component implementing the SSO authentication mechanism.

Figure 10. Configurator metamodel



For response time and availability, the components are annotated, at runtime, for each tenant that requests the service with the value that satisfies its required quality levels.

However, if the monitoring system detects SLA violation when checking SLA non-conformance for each tenant, then it will detect which tenants are concerned with this violation and notifies the Configurator about this state to take adequate actions.

For example, if the measured response time metric for the application is 600ms, the monitoring system will alert the configurator about policies violation for tenant1 and tenant2 as their respective required values for response time are 500ms and 300ms. To cope with that, the Configurator will perform the scale action. This action will rely on the Cloud offering, Infrastructure as a Service or Platform as a Service, the SaaS application is built on. Generally, this offering provides for SaaS providers mechanisms to ensure rapid elasticity by enabling automatic adding and removing resources depending on the load. Therefore, the Configurator component will rely on these means to perform rapidly its actions that relate to the underlying infrastructure of the SaaS service.

In this paper, the middleware is presented, with its Policies Generator and Configurator components in detail. The middleware presumes that the SaaS application is based on SCA to deal with the application as an assembly of components. The focus was on Policy Generator and Configurator components of the middleware, while the rules engine and the monitoring system will be addressed in the ongoing work.

DISCUSSION AND EVALUATION

In this section, the proposed approach will be evaluated against the non-functional variability challenges mentioned in Section 2, and against the criteria for product-line implementation techniques defined in (Apel & Batory, 2013).

Non-Functional Variability Challenges

Challenge 1: Modeling Non-Functional Requirements (NFR) and Their Variability

The approach has dealt with this challenge through the Domain Engineering phase. The NFVariability metamodel is defined to model the variability in NFR in a high level of abstraction so as to widen the scope of the supported quality attributes. The dependencies between the NFRs are also supported, i.e. when an NFR requires another or excludes it. This part has been thoroughly treated in ((Aouzal et al., 2018).

Challenge 2: Deriving Tenant-Specific Policies

This challenge is addressed through model-to-model transformations from the Generic SLA, the instance of the VariabileSLA metamodel, to core and tenant-specific policies, instances of the Policy metamodel. Therefore, the authors adopted a policy-driven adaptation by considering the policies as the driver of each needed configuration.

Challenge 3: Configuring the SaaS Service Dynamically According to Quality Attributes at Runtime

This challenge is addressed through performing dynamic configurations by injecting annotations into the components at runtime. It is addressed also by establishing a configuration cycle through which the context is monitored and the Configurator is informed about context changes. This is partially dealt with in the current paper, by introducing the Event interpreter in the Configurator metamodel. It represents the change occurred that is translated to an adequate action to make the adaptation at runtime.

SPL Criteria

These criteria as defined in (Apel & Batory, 2013) are: Low preplanning effort, feature traceability, separation of concerns, information hiding, granularity, and uniformity:

- **Low preplanning effort:** Preplanning aims at anticipating the changes and easing the reuse and variability mechanisms. It is an incurred aspect in SPL engineering (Apel & Batory, 2013). This preplanning effort is basically made, in the proposed approach, at design time and during the Domain Engineering process when documenting the domain requirements, mapping them to mandatory and optional features, and anticipating the modeling of the intra- and inter-dependencies of the existing features with new features integrated at runtime. The preplanning effort is considerably tedious in the first phase at design time, but this effort is paid off by the ease of reuse at runtime and it is done once at the Domain Engineering phase. Moreover, the architectural styles the SaaS application is based on reduce considerably this effort as they implement mechanisms, such as loose coupling and composition, which foster reusability and incorporating new features in a seamless way by reusing existing components;
- **Feature Traceability:** This criterion concerns the mapping between the problem space (Domain Engineering and its subsequent models) and the solution space (Application Engineering and the implementation of the models of the problem space). Ensuring feature traceability for quality attributes seems difficult as some quality attributes do not have a concrete implementation and they are satisfied depending on functional components, other non-functional properties or infrastructure metrics. However, feature traceability is satisfied in the proposed approach at some extent thanks to the use of MDE and ETL as the transformation language. ETL allows elaborating additional model (Kolovos et al., 2018) that preserves the trace of the features from NFR in the EFM to annotations passing by SLA terms and policies;
- **Separation of concerns:** Separating concerns, as defined in (Apel & Batory, 2013), refers to the ability to separate features into distinct and cohesive artifacts in order to facilitate the maintenance and evolution of a feature. As the proposed approach is annotation-based, the features crosscut the code as annotations and this hinders the principle of separation of concerns on the point of view of SPLE. However, the separation of concerns is achieved and simulated by the configurator as it is the "tool" that annotates the service components with quality attributes and maintains them and supervises their evolution. Moreover, defining core and tenant-specific policies responds to this criterion by separating functionality from non-functional aspects and it is enforced by the SCA architecture that helps implementing cohesive components;
- **Information hiding:** It represents the decomposition of a module into internal and external parts (Apel & Batory, 2013). The information related to the implementation techniques, for example, is hidden in the internal part whereas the external part exposes the functionality of the module to other modules. In the proposed approach, this is supported by the SCA architecture that enables exposition of the functionality of each component through interfaces. Regarding the non-functional features, the information is hidden in the annotations fragments, which refer to specific implementations, and what is exposed is the quality levels that the component should ensure;
- **Granularity:** It refers to the level of granularity on which the variability features are implemented. It can be coarse-grained, e.g. at the level of classes, medium grained at the level of member classes for example, or fine-grained by modifying, for example, an instruction in a method body (Apel & Batory, 2013). In the proposed approach, non-functional variability is implemented as annotations. Since annotations can operate and be applied at the level of small pieces of the code base, annotation-based implementations support fine-grained variability. Coarse-grained variability can be achieved by exploiting the benefits of SCA as being a composition-based architecture which enforces coarse-grained and medium-grained variability at the level of components (Apel & Batory, 2013; Horcas, Cortiñas, & Luaces, 2018);

- **Uniformity:** The principle of uniformity refers to the fact that all artifacts, be it annotated or composed, should be encoded and synthesized in a similar way, though they are of different kinds that include code and noncode artifacts (Apel & Batory, 2013). As the proposed approach relies on MDE principles which advocate building language-independent and platform-independent tools, the principle of uniformity is achieved as the variability artifacts are not defined in a specific technology and their interaction is not hindered by a specific language or implementation.

RELATED WORK

In this section, the authors present related work regarding two aspects: the first one is variability management in multi-tenant SaaS applications and the second one is non-functional variability management:

- **Variability management in multi-tenant SaaS applications:** Most of the approaches dealing with adaptation in multi-tenant SaaS applications involve SPLE techniques combined with other paradigms. Regarding SPLE techniques, Feature modeling is used extensively in most works, e.g. (Landuyt et al., 2015; Tizzei et al., 2017), as a key technique to capture variability in multi-tenant SaaS applications during domain analysis. In some works, Feature model was used as a design time artifact like in (Cao et al., 2015; Etedali et al., 2017) where features are encoded and modeled in XML format to tailor them with the cloud configuration that meets the tenants' requirements. Feature modeling is also combined with MDE techniques to model variability in Cloud services. In (Abu-Matar et al., 2014), the authors proposed a framework based on a set of meta-views and views to represent the requirements and the architecture aspects of a Cloud service. With the same MDE perspective, the work (Mohamed et al., 2017) proposed an integrated platform that supports both Feature-level variability management and runtime variability management. The use of MDE in their work is motivated by the high level of abstraction which enables managing customization in a better way. With the emergence of Dynamic Software Product Lines (DSPL) (Hinchey et al., 2012), Feature model is used as a runtime artifact. In (Gey et al., 2014), the authors enabled this concept of runtime Feature model by relying on models at runtime techniques. They designed a feature middleware that ensures dynamic adaptation of multi-tenant SaaS applications through querying Feature bundles. Runtime adaptation is also achieved through the use of self-adaptation MAPE (Monitoring, Analysis, Provisioning and Execution) loop. In (García-galán et al., 2016; García-Galán et al., 2014), the authors define a user-centric adaptation based on MAPE loop that aims at choosing the configuration, from the solution space, that maximizes tenants' preferences.

The abovementioned works focus mainly on the functional aspect of variability. Their solutions deal with configuring and adapting the SaaS applications to functional requirements of the tenants and their non-functional requirements are completely or partially discarded.

- **Non-functional variability management:** Some approaches deal with non-functional variability alongside with functional variability. In (Bagheri & Du, 2017), the authors detail the application engineering process for configuring functional features alongside with non-functional ones. Regarding non-functional properties, they consider them as independent elements and they tackle them using the concept of soft-goal to quantify the impact of a functional feature on a quality attribute. Other approaches focus on variability of a specific quality attribute. The work presented in (Myllärniemi et al., 2016) deals with purposeful performance variability. The authors of this work proposed general theoretical models to study and understand the motivations beyond performance variability and the strategy to adopt to realize this variability in the product line architecture. In the context of self-adaptive systems, the works (Tamura et al., 2010; Tamura et al., 2013; Villegas et al., 2011) proposed QoS contract-aware approaches to assess the adaptation properties and to reconfigure dynamically the systems in case of QoS contract violations to preserve the expected quality levels. Self-adaptive systems principles are adopted in (Gey et al., 2016) to design a middleware responsible for continuous evolution of multi-tenant SaaS applications according to SLAs of their tenants while involving the tenants in this process. Self-adaptation is supported in (Psannis et al., 2018), where the authors developed an algorithm that selects and streams the video that matches what the user of an Intelligent Cloud environment has demanded in terms of quality. Within the scope of multi-tenant service-based systems, the authors in (Wang, He, Ye, & Yang, 2017) proposed a criticality-based fault tolerance strategy that aims at replicating the critical component services. The criticality of a component is measured based on its multi-dimensional quality and the tenants that share that component. Considering the dynamicity of the Cloud environment as the key trigger of change and adaptation, the work in (Rafique et al., 2019) defines SCOPE, a middleware that encompasses self-adaptive capabilities for data management in federated clouds with the aim of autonomous (re)configuration of the underlying storage architecture.

To sum up, variability management is extensively dealt with in multi-tenant SaaS applications. However, the focus is on the functional aspect of variability. Some approaches handled variability in quality attributes, but they are limited to only some of them, e.g. performance, or they are applied beyond multi-tenant SaaS applications. Unlike these works, the approach proposed in this paper focuses on non-functional variability, and is an integrated approach that spans the Domain Engineering and Application Engineering phases, that is generic and not dependent to specific quality requirements, and that provides means for runtime configuration to maintain the evolution of the services according to the policies required by the tenants.

As the proposed approach deals with the non-functional variability at the application level, it must consider the quality attributes that are tied to the underlying infrastructure of the cloud environment. At the current state of the proposed middleware, the monitoring system is envisioned in order to communicate the infrastructure state to the application and to ensure autonomous reconfiguration. Nevertheless, the monitoring system by itself is not sufficient; there is a prominent need to explore other mechanisms such as service brokerage and intelligent datacenters allocation in order to route the user request to the datacenter that best matches the user and infrastructure constraints (Manasrah et al., 2017).

CONCLUSION AND FUTURE WORK

Building SaaS applications under the multi-tenancy architecture brings many advantages in terms of cost and maintenance, and leverages largely the economies of scale. This architectural style enables sharing one service instance among multiple tenants. Hence, to keep tenants' specificities, the service should be configurable and the induced variability should be addressed. In this paper, the authors deal with non-functional variability. This variability refers to the fact that the quality levels differ from a tenant to another. This gives rise to SLA violations if not handled correctly. Therefore, this paper proposes an integrated approach that adopts the principles of SPLE and MDE. It proposes a middleware that configures SaaS service components based on policies derived from non-functional requirements through model-to-model transformations using ETL as the transformation language. The approach allows configurations at design time and runtime through annotating the service components with adequate quality levels.

As future work, the authors will provide the middleware, especially the Configurator component, with means for conflicting quality attributes optimization in order to define a trade-off and to choose the optimum solution that best satisfies the conflicting objectives. Moreover, they intend to define a microservices-based architecture for SaaS applications in order to have independent and self-contained components and for better reuse strategy. The monitoring system will also be defined in the on-going work. The proposed middleware along with the monitoring system will help in keeping the quality levels of SaaS services as expected by the tenants and reducing SLA violations.

REFERENCES

Abu-Matar, M., Mizouni, R., & Alzahmi, S. (2014). Towards Software Product Lines Based Cloud Architectures. In *2014 IEEE International Conference on Cloud Engineering* (pp. 117–126). IEEE. doi:10.1109/IC2E.2014.10

Aouzal, K., Hafiddi, H., & Dahchour, M. (2018). Handling Tenant-Specific Non-Functional Requirements through a Generic SLA. In *13th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2018)*, 383 -- 391.

Apel, S., & Batory, D. (2013). *Feature-Oriented Software Product Lines*. Springer.

Bagheri, E., & Du, W. (2017). Toward automated quality-centric product line configuration using intentional variability. *Journal of Software: Evolution and Process*, (March), 1–26. doi:10.1002/smr.1870

Batory, D. (2005). Feature Models, Grammars, and Propositional Formulas. In *Software Product Line Conference* (pp. 7–20). doi:10.1007/11554844_3

Benavides, D., Trinidad, P., & Ruiz-Cortés, A. (2005). *Automated Reasoning on Feature Models. In 17th International Conference on Advanced Information Systems Engineering (CAiSE'05)* (pp. 491–503)., Retrieved from http://link.springer.com/10.1007/11431855_34

Cao, Y., Lung, C., & Ajila, S. (2015). Constraint-based Multi-tenant SaaS Deployment Using Feature Modeling and XML Filtering Techniques. In IEEE 39th Annual International Computers, Software & Applications Conference. IEEE. doi:10.1109/COMPSAC.2015.255

Etedali, A., Lung, C., Ajila, S., & Veselinovic, I. (2017). Automated Constraint-based Multi-tenant SaaS Configuration Support Using XML Filtering Techniques. In *IEEE 41st Annual Computer Software and Applications Conference*. IEEE. doi:10.1109/COMPSAC.2017.69

García-Galán, J., Pasquale, L., Trinidad, P., & Ruiz-Cortés, A. (2014). User-centric Adaptation of Multi-tenant Services: Preference-based Analysis for Service Reconfiguration. In *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems* (pp. 65–74). doi:<ALIGNMENT.qj></ALIGNMENT>10.1145/2593929.2593930

García-galán, J., Pasquale, L., Trinidad, P., & Ruiz-Cortés, A. (2016). User-Centric Adaptation Analysis of Multi-Tenant Services. *ACM Transactions on Autonomous and Adaptive Systems*, *10*(4), 1–26. doi:10.1145/2790303

Gey, F., Van Landuyt, D., & Joosen, W. (2016). Evolving multi-tenant SaaS applications through self-adaptive upgrade enactment and tenant mediation. In *Proceedings of the 11th International Workshop on Software Engineering for Adaptive and Self-Managing Systems - SEAMS '16* (pp. 151–157). doi:10.1145/2897053.2897057

Gey, F., Van Landuyt, D., Walraven, S., & Joosen, W. (2014). Feature Models at Run Time Feature Middleware for Multi-tenant SaaS Applications. In *Proceedings of the 9th Workshop on Models@run.time co-located with 17th International Conference on Model Driven Engineering Languages and Systems MODELS,* Valencia, Spain (pp. 21--30). CEUR-WS.org. Retrieved from http://ceur-ws.org/Vol-1270

Guerra, E., Cardoso, M., Silva, J., & Fernandes, C. (2010). Idioms for code annotations in the Java language. In *Proceedings of the 8th Latin American Conference on Pattern Languages of Programs - SugarLoafPLoP '10* (pp. 1–14). doi:10.1145/2581507.2581514

Hinchey, M., Park, S., & Schmid, K. (2012). Building Dynamic Software Product Lines. *IEEE Computer*, (10), 5–10.

Horcas, J., Cortiñas, A., & Luaces, M. R. (2018). Integrating the Common Variability Language with Multilanguage Annotations for Web Engineering. In SPLC '18. doi:10.1145/3233027.3233049

Jennings, B., & Stadler, R. (2014). Resource Management in Clouds: Survey and Research Challenges. *Journal of Network and Systems Management*, *23*(3), 567–619. doi:10.1007/s10922-014-9307-7

Kabbedijk, J., Bezemer, C., Jansen, S., & Zaidman, A. (2015). Defining Multi-Tenancy: A Systematic Mapping Study on the Academic and the Industrial Perspective. *Journal of Software and Systems, 100*, 139-148.

Kolovos, D., Rose, L., Paige, R., & Garcıa-Domınguez, A. (2010). The epsilon book. *Structure (London, England)*, *178*, 1–10.

Manasrah, A. M., Aldomi, A., & Gupta, B. B. (2017). An optimized service broker routing policy based on differential evolution algorithm in fog/cloud environment. *Cluster Computing*, 1–15. doi:10.1007/s10586-017-1559-z

Mohamed, F., Mizouni, R., Abu-Matar, M., Al-qutayri, M., & Whittle, J. (2017). An Integrated Platform for Dynamic Adaptation of Multi-Tenant Single Instance SaaS Applications. In *IEEE 5th International Conference on Future Internet of Things and Cloud*. doi:10.1109/FiCloud.2017.39

Myllärniemi, V., Savolainen, J., Raatikainen, M., & Männistö, T. (2016). Performance variability in software product lines: Proposing theories from a case study. *Empirical Software Engineering*, *21*(4), 1623–1669. doi:10.1007/s10664-014-9359-z

Nosál, M., & Porubän, J. (2014). XML to Annotations Mapping Definition with Patterns. *Computer Science and Information Systems*, *11*(4), 1455–1477. doi:10.2298/CSIS130920049N

OASIS. (2011). Service Component Architecture Assembly Model Specification Version 1.1 Committee Specification Draft 09 / Public Review Draft 04.

Psannis, K., Stergiou, C., & Gupta, B. B. (2018). Advanced Media-based Smart Big Data on Intelligent Cloud Systems. *IEEE Transactions on Sustainable Computing*, *3782*(c), 1–1. doi:10.1109/tsusc.2018.2817043

Rafique, A., Van Landuyt, D., Truyen, E., Reniers, V., & Joosen, W. (2019). SCOPE: Self-adaptive and policy-based data management middleware for federated clouds. *Journal of Internet Services and Applications*, *10*(1), 2. doi:10.1186/s13174-018-0101-8

Tamura, G., Casallas, R., Cleve, A., & Duchien, L. (2010). QoS Contract-Aware Reconfiguration of Component Architectures Using E-Graphs. In *7th International Workshop on Formal Aspects of Component Software*.

Tamura, G., Casallas, R., Cleve, A., & Duchien, L. (2013). QoS contract preservation through dynamic reconfiguration: A formal semantics approach. *Science of Computer Programming*, *94*(P3), 307–332. doi:10.1016/j.scico.2013.12.003

Tizzei, L. P., Nery, M., Segura, V. C. V. B., & Cerqueira, R. F. G. (2017). Using Microservices and Software Product Line Engineering to Support Reuse of Evolving Multi-tenant SaaS. In *Proceedings of the 21st International Systems and Software Product Line Conference* - Volume A *on - SPLC '17* (pp. 205–214). doi:10.1145/3106195.3106224

Van Der Linden, F., Bosch, J., Kamsties, E., Känsälä, K., & Obbink, H. (2004, August). Software product family evaluation. In International Conference on Software Product Lines (pp. 110-129). Springer. doi:10.1007/3-540-28901-1

Van Landuyt, D., Walraven, S., & Joosen, W. (2015). *Variability Middleware for Multi-tenant SaaS Applications* (pp. 211–215). SPLC.

Villegas, N. M., Müller, H. A., Tamura, G., Duchien, L., & Casallas, R. (2011). A framework for evaluating quality-driven self-adaptive software systems. In *Proceeding of the 6th international symposium on Software engineering for adaptive and self-managing systems - SEAMS '11*. doi:10.1145/1988008.1988020

Wang, Y., He, Q., Ye, D., & Yang, Y. (2017). Formulating Criticality-Based Cost-Effective Fault Tolerance Strategies for Multi-Tenant Service-Based Systems. *IEEE Transactions on Software Engineering, 44(3), 291-307.* doi:10.1109/TSE.2017.2681667

Khadija Aouzal is currently a PhD student in Computer Science and Engineering at the National Institute of Posts and Telecommunications, INPT, Rabat, Morocco. She received her Eng. Degree in Computer Science from INPT. Her research interests include variability management in Cloud Computing and dynamic adaptation of services regarding non-functional requirements of users.

Hatim Hafiddi is a Professor of Software Engineering at the National Institute of Communication (INPT), Rabat, Morocco. He holds an Engineer degree and a PhD degree in Software Engineering from the National College of IT (ENSIAS), Rabat, Morocco. Hatim has more than five years of experience in the software development industry. His research interests are context-aware service-oriented computing, mobile information systems engineering and IS governance in cloud computing.

Mohamed Dahchour received a doctorate degree (2001) in computer science from Ecole polytechnique de Louvain, Université catholique de Louvain, Belgium. Currently, he is a full professor of computer science at the National Institute of Posts and Telecommunications (INPT), Morocco. His scientific interests include software engineering, conceptual modeling, web semantic, distributed systems, and information systems management.