Handling Tenant-Specific Non-Functional Requirements through a Generic SLA

Khadija Aouzal[1], Hatim Hafiddi[1,2] and Mohamed Dahchour[1]
[1]*INPT, Rabat, Morocco*
[2]*ENSIAS, Rabat, Morocco*

Keywords: SaaS, Non-Functional Variability, SPLE, MDE, QoS Characteristics, SLA.

Abstract: In a multi-tenant architecture of a Software as a Service (SaaS) application, one single instance is shared among different tenants. However, this architectural style supports only the commonalities among tenants and does not cope with the variations and the specific context of each tenant. These variations concern either functional or non-functional properties. In this paper, we deal with non-functional variability in SaaS services in order to support the different quality levels that a service may have. For that purpose, we propose an approach that considers Service Level Agreements (SLAs) as Families in terms of Software Product Line Engineering. We define two metamodels: NFVariability metamodel and VariableSLA metamodel. The first one models and captures variability in quality attributes of services. The second one models a dynamic and variable SLA. Model-to-model transformations are performed to transform Feature Model (NFVariability metamodel instance) to Generic SLA (VariableSLA instance) in order to dynamically deal with the tenant-specific non-functional requirements.

1 INTRODUCTION

Software as a service (SaaS) is a software delivery model, which represents the capability offered to the customer to use, on an on-demand and a pay-as-you-go bases, the provider's application hosted on a cloud infrastructure (Mell and Grance, 2011).

SaaS applications are mainly hosted based on multi-tenancy, in order to maximize the advantages of economy of scale (Bezemer and Zaidman, 2010). In a multi-tenant architecture, one single instance of the application is shared among different tenants, which enables cost efficiency and easy maintenance for the providers. However, this architectural style, based on one-size-fits-all approach, supports only the commonalities among the tenants and does not cope with the variations and the specific context of each tenant. These variations are introduced at two main and major levels: at the level of functionality in terms of e.g. variant business logics, service compositions, etc; and at the level of quality attributes in terms of e.g. performance, availability, security, etc. This variability is driven by several aspects such as law regulations, nature of processed data, tenant-specific requirements, etc. Therefore, variability is an inherent aspect that characterizes a given service and which crosscuts all the layers of a SaaS application: Presentation, Business, Service and Data layers. Due to these reasons, the tenants need to have services tailored to their particular needs, especially when it comes to non-functional requirements in a multi-tenant environment. Our focus is on non-functional variability at the level of the service layer.

Variability management is a core concept of Software Product Lines Engineering (SPLE) (Pohl et al., 2005). This paradigm enables high reusability of software artefacts by building product families or lines and then deriving the final product tailored to specific contexts or users. A product family contains members that share a set of commonalities and that have variabilities which characterize each one of them. These commonalities, variabilities and the relationships between them are usually modeled using Feature Model (FM) (Batory, 2005) or Orthogonal Variability Model (OVM) (Pohl et al., 2005) which are structured as hierarchical trees. In the Feature Model, both commonalities and variabilities are identified, respectively, as mandatory and optional features. However, in the Orthogonal Variability Model, only the variabilities are modeled exploiting, from SPLE, the concepts of variation point and variant which refer to a variable item and to the instantiation of this variable item, respec-

Aouzal, K., Hafiddi, H. and Dahchour, M.
Handling Tenant-Specific Non-Functional Requirements through a Generic SLA.
DOI: 10.5220/0006791403830391
In *Proceedings of the 13th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2018)*, pages 383-391
ISBN: 978-989-758-300-1

tively. In the proposed approach, we use the bases and concepts of Feature Model to express and represent variability in Non-Functional Properties of SaaS services.

Since quality level of services is specified through the Service Level Agreement (SLA) contract, this paper proposes an approach that considers SLAs as Families in terms of Software Product Line Engineering. We introduce variability in SLA through the notion of Generic SLA, a document that encompasses the terms of all contracting tenants. We define two metamodels: NFVariability metamodel and VariableSLA metamodel. The first one, which is based on the QoS Metamodel of QoS&FT profile (OMG, 2008), models and captures variability in quality attributes of services. The second one models a dynamic and variable SLA. Model-to-model transformations are performed to transform Feature Model (NFVariability metamodel instance) to Generic SLA (VariableSLA instance).

The remainder of the paper is structured as follows. Section 2 gives generalities about SLA and QoS Framework Metamodel of QoS&FT standard. Section 3 presents related works. Section 4 presents the problem statement. Section 5 describes the proposed metamodels and the process for generating Generic SLA. Section 6 concludes the paper and gives future work for our on-going research.

2 BACKGROUND

2.1 Service Level Agreement

Service Level Agreement (SLA) is a contract established between service provider and service consumer. It aims to clearly define and monitor the service and its quality requirements. Generally, its clauses cover: the involved parties which are, in most cases, the service provider and the service consumer; the terms agreed upon after negotiation; and penalties against the service provider in case of SLA violations. Several languages have been proposed to define SLAs, namely: WS-Agreement (Andrieux et al., 2006), WSLA (Keller and Ludwig, 2003), SLAC (Uriarte et al., 2014) and CSLA (Serrano et al., 2016). The two former languages are used to specify SLAs in Web Services, as for the two latter ones, they are used for Cloud services. In this paper, we introduce a novel concept and structure of SLA, Generic SLA, which supports variability of quality attributes across multiple tenants. This concept goes in line with SPLE principles, as it considers commonalities and variabilities of quality attributes, and forms a family of tenant-specific SLAs derived from the Generic SLA.

2.2 QoS Metamodel of QoS&FT Profile

The QoS&FT profile comprises two metamodel frameworks: QoS Metamodel and Fault Tolerance Metamodel (OMG, 2008). In our approach, we are interested in QoS Metamodel as it models quality characteristics in software systems. This QoS framework metamodel comprises three packages which represent three metamodels: QoSCharcteristics metamodel, QoSConstraints metamodel and QoSLevels metamodel. The QoSCharacteristics metamodel consists of modeling quantifiable quality characteristics of services and their dimension of quantification. The QoSConstraints metamodel introduces concepts of constraints over service qualities, QoS offered, QoS required and QoS contract. As for QoSLevels metamodel, it defines levels of qualities that depend on different execution modes of the system modeled.

3 RELATED WORKS

Variability management has been subject of many researches in different domains. Recently, the interest was on extrapolating these works to the area of Cloud Computing. Therefore, many researches are conducted in order to build Cloud applications tailored to customers'requirements, however the emphasis was on functional requirements (Aouzal et al., 2015).

In (Fehling et al., 2011), the authors introduced a framework that enables customization and provisioning of flexible cloud applications. It provides a self-service portal to end-users in order to configure the application according to their needs. This framework, however, does not support multi-tenancy and non-functional variability. In (Abu-Matar et al., 2014), the authors model variability in Cloud services using SPL techniques, multi-view modeling and MDE approaches. The proposed framework is constituted of a set of meta-views and views that describe cloud services in their requirements and architecture aspects. Considering users' preferences changes as the main driver of adaptation, the approach presented in (García-Galán et al., 2014; García-galán et al., 2016) relies on the activities of MAPE (Monitoring, Analysis, Provisioning and Execution) loop to adapt shared multi-tenant services in a user-centric manner in order to maximize users'satisfaction. In order to ensure dynamic adaptation and configuration of multi-tenant SaaS applications, a feature middleware is designed (Gey et al., 2014) as a runtime artifact. In (Landuyt et al., 2015), the authors presented, through the notion of service lines, a middleware that supports variability in multi-tenant SaaS applications, with a focus on the operational aspects. In (Tizzei et al., 2017) the authors rely on SPLE and microservices to support both reuse and independent evolution of multi-tenant SaaS services.

However, the aforementioned works mostly focus on adapting multi-tenant services according to tenant functional needs, and they do not deal with non-functional variability, its modeling and the dependencies between quality attributes.

A systematic literature review conducted on variability in software systems (Galster et al., 2014) concluded that the focus on variability in Quality Attributes is minor and the works were mostly interested in specific Quality Attributes, such as: performance, availability, security, etc. To cope with that, (Galster, 2015) gives research directions regarding variability in Quality Attributes. In (Horcas et al., 2016; Horcas et al., 2017), the authors rely on SPLE approach to model Functional Quality Attributes (FQAs) in software applications, and on Aspect Oriented Programming to inject FQAs into the application. Contrary to that work, our approach is generic and not limited to FQAs.

To the best of our knowledge, no work has considered SLA as a product family, though several SLA languages were proposed to specify and define Cloud services quality such as (Boukadi et al., 2016; Serrano et al., 2016; Uriarte et al., 2014; Tata et al., 2016; Mohamed et al., 2017).

4 PROBLEM STATEMENT

4.1 Non-Functional Variability

Non-functional variability in service-based systems, namely SaaS applications, is defined in (Mahdavi-Hezavehi et al., 2013) as the ability of a service to be tailored to the different non-functional requirements of tenants, thus offered with several levels of non-functional properties. Non-functional variability is classified in (Galster, 2015) into two categories, according to the sources that trigger it and introduce the variations: intentional variability and unintentional variability. Intentional variability is directly tied to the variations of non-functional requirements of tenants. As for unintentional variability, it is introduced due to variations in hardware resources, and due to the interactions between functional properties and non-functional ones and within non-functional properties themselves.

Therefore, non-functional variability introduces several challenges in building configurable and customizable multi-tenant SaaS services; particularly, the issue of modeling variability in non-functional requirements and linking it with service quality level specified in SLA.

4.2 Motivating Scenario

We illustrate the problem with a document management application, which serves multiple tenants using the SaaS model. This application automates the activities of creation, management and storage of any nature of documents: contracts, financial documents, etc. It gives tenants and end users control and visibility on modifications made to documents or contracts, 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 real time and accurate.

Since the application is implemented following the multi-tenancy architecture, i.e. a single application instance serves multiple tenants, ensuring security is a crucial need. Therefore, in terms of security, the application requires, among others: 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) and 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 or/and the user location.

The application has to cater for variant levels of non-functional properties that characterize each tenant. For instance, consider three tenants that consist of: Medical Insurance Provider, Healthcare Analytics Provider and a Research Laboratory. The Medical Insurance Provider uses the application to manage his documents and contracts with existing and new clients. Concerning security, he requires authentication, data integrity and privacy and user access controls. He requires also an availability of 99.95%, a response time not exceeding 500 ms, and the service to be device and location-aware. As for the Healthcare Analytics Provider, he makes use of the application as a central repository where he manages his contracts with all his clients, and where he has visibility on the status of contracts all along their cycle, from negotiation to signature. His security requirements are the same as for the first tenant. However, he differs in availability and response time; he requires 99% and 300 ms, respectively; and he does not require device awareness. The third tenant automates the process of research papers review through the application. For this purpose, the research laboratory requires authentication, privacy, user access controls and session duration and time management; as well as 98% and 800 ms of availability and response time, respectively. But, he does not want the service to consider adaptability.

Table 1 summarizes these requirements for the three tenants.

5 GENERIC SLA GENERATION PROCESS

5.1 Approach Overview

In order to have a SaaS service that meets non-functional requirements of tenants, we believe that an initial step for that purpose would be to model non-functional variability in SaaS services. Therefore, our approach combines MDE and SPLE principles. Figure 1 depicts the process for generating Generic SLA.

First, variations in non-functional requirements are represented using Feature Models. These models are in accordance with NFVariability Metamodel. This metamodel, intends to model non-functional properties of SaaS services, their dependencies, and their variable aspect, i.e. different levels of a QoS attribute for the same service. Therefore, QoS characteristics are represented, in this metamodel, as variable features that can be either mandatory or optional. After instantiation of this metamodel to Feature Model, model-to-model transformations are performed in order to generate a Generic SLA. The generic SLA is a global and dynamic SLA that contains all the non-functional properties of all tenants, .i.e. an SLA with variability represented at the level of its terms and its Service Level Objectives (SLOs).

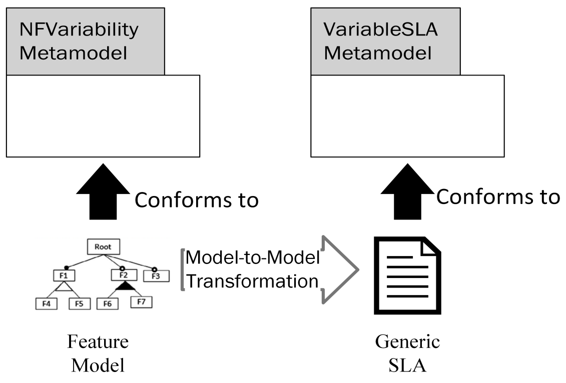

Figure 1: Generic SLA generation process.

It is in conformance with VariableSLA metamodel, which represents variability in SLA.

5.2 NFVariability Metamodel

Figure 2 depicts the proposed metamodel that extends the QoS Metamodel of QoS&FT profile with variability concepts.

In this metamodel, a capability represents the SaaS service offered to tenants. Each capability can be associated to multiple QoS characteristics. QoS characteristics represent quantifiable non-functional requirements, grouped into categories such as: performance category to denote availability, response time, reliability, etc; or security category to represent authentication, authorization, access control, integrity, etc. QoS characteristics are modeled as features that may be either mandatory or optional. A mandatory QoS attribute means it is selected for all the tenants. An optional one may be selected for just the tenants that required it. Variation Point and Variant in QoS characteristics are represented, respectively, by the references parent and child. QoS characteristics that have the same feature parent, i.e. characteristics that are brothers, can rely on each other through these three operators: AND, OR and Alternative. This relation is modeled by Association. An AND operator between two quality attributes means that those AND-linked attributes may be selected in the derived application, depending on their optionality. Two OR-linked quality attributes mean that one or both of them may be selected. The Alternative operator plays the role of XOR, which means the presence of a quality attribute variant excludes the presence of another

Table 1: Non-functional requirements of different tenants.

Tenant	Security				Performance		Adaptability	
	Integrity, Privacy, Access Control	Session Duration and Time Mngt	Authentication		Availability	Response Time	Device Awareness	Location Awareness
			Login Credentials	QR Code				
Medical Insurance Provider	+	-	+	-	99.95%	500ms	+	+
Healthcare Analytics Provider	+	-	+	-	99%	300ms	-	+
Research Laboratory	+	+	+	+	98%	800ms	-	-

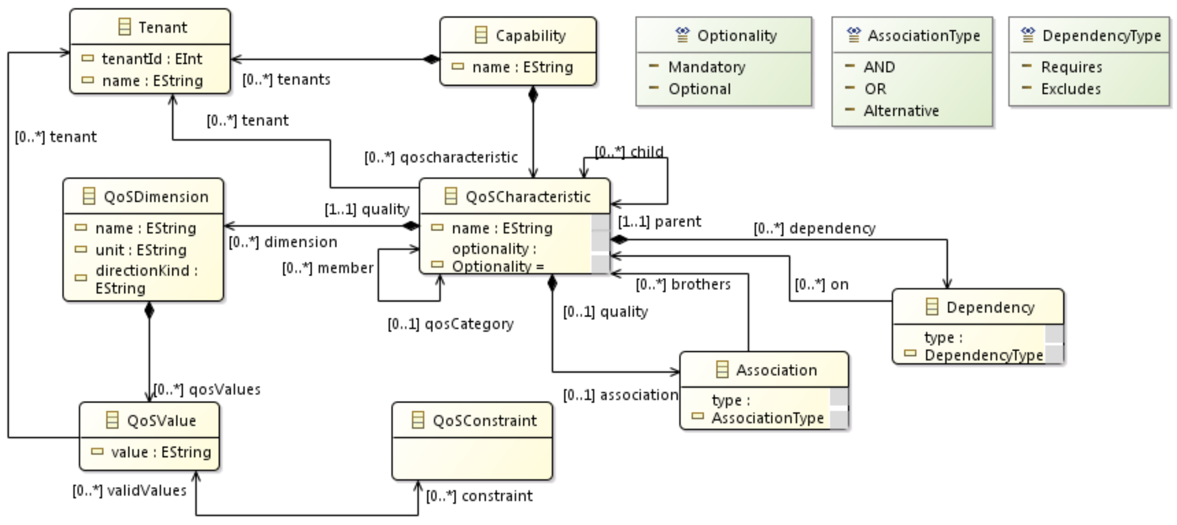

Figure 2: NFVariability Metamodel for modeling variability in quality attributes.

variant. The interdependencies between QoS characteristics variants are modeled by Dependency using two types: requires and excludes. QoSDimension defines the different ways a QoS characteristic can be quantified, e.g. for response time we can consider execution time or network time or both (Boukadi et al., 2016). A QoS dimension can have different values, and each value can refer to multiple tenants. QoSConstraint models the constraints, e.g. constraints introduced by the infrastructure, which limit the space of valid values for a certain QoS dimension.

The following OCL constraints define some rules that enhance the metamodel semantic:

1. Brothers definition:

```
Context QoSCharacteristic :: brothers :
Set(QoSCharacteristic)
derive : self.child -> union(self.child.
    brothers)
```

2. An optional QoS characteristic cannot exclude a mandatory one:

```
context QoSCharacteristic inv:
self.optionality = optional implies self.
    association -> forAll(a | a.brothers.
    optionality = mandatory) -> including (
    self.association.type = Excludes) ->
    isEmpty()
```

3. When a parent feature is mandatory, at least one of its children should be mandatory:

```
context QoSCharacteristic inv:
self.parent.optionality = mandatory implies
    select ( self.child.optionality =
    mandatory) -> size() = 1
```

4. AND-linked features can be either mandatory or optional

```
context Association inv:
self.type = AND implies ((self.quality.
    optionality = mandatory or self.quality.
    optionality = optional) and
(self.brothers.optionality= mandatory or
self.brothers.optionality=optional)
```

5. The alternative association is not allowed between two manadatory variants as they both must exist in the application:

```
context QoSCharacteristic inv:
self.association.type = Alternative implies
    forAll(b | b.brothers.optionality <>
    mandatory)
```

After domain analysis of an application and extraction of its non-functional requirements, non-functional variability is documented in accordance to the NFVariability Metamodel. This variability is modeled using Feature Models as illustrated in figure 3 for Document Management application. This feature model contains also some cross-tree constraints that represent quality attributes interdependencies, and they denote respectively: 1) Session Duration requires Access Control; 2) Standard Cost excludes any optional quality attribute, as this cost concerns only basic and mandatory attributes; 3) Premium Cost requires at least one optional feature.

5.3 VariableSLA Metamodel

The VariableSLA Metamodel is shown in figure 4. It represents the contract SLA that supports variabil-

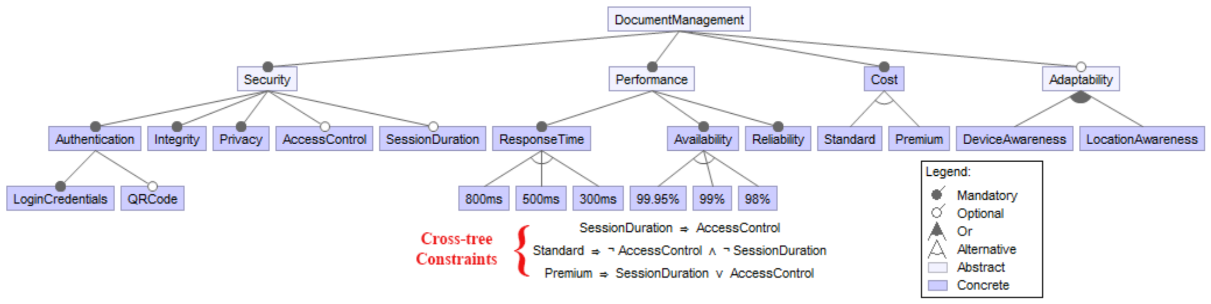

Figure 3: Feature Model for Document Management Service.

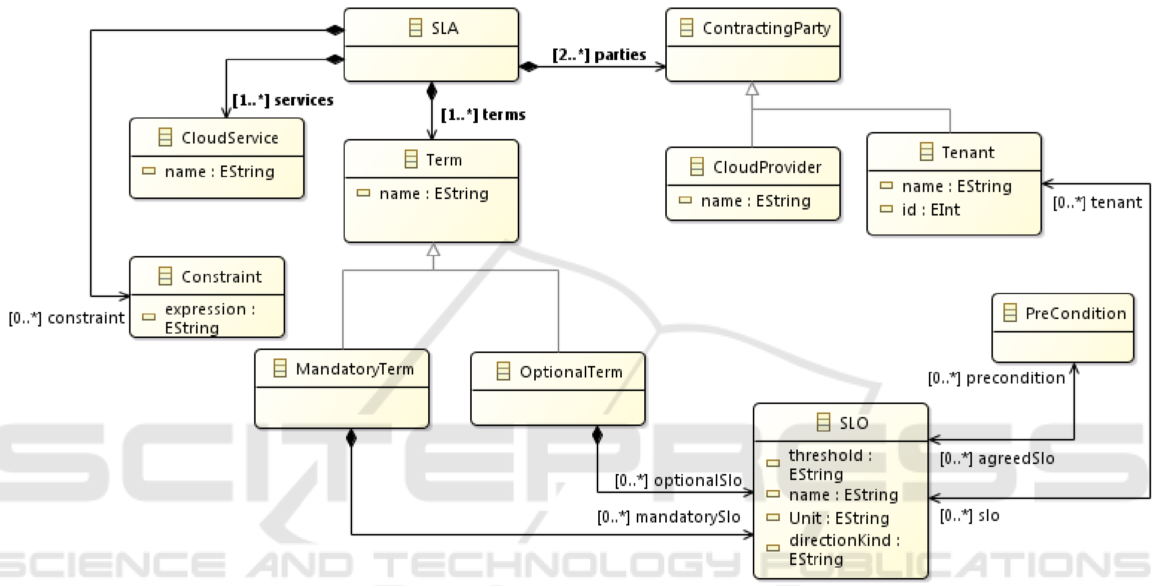

Figure 4: VariableSLA Metamodel for representing variability in SLA.

ity. It is structured into four main blocks: Contracting Parties, Cloud Services, Terms and Constraints.

- Contracting Parties define the actors involved in the contract: Cloud Provider and Tenant (Cloud Consumer).
- Cloud Services represent the scope of services covered by the agreement.
- Terms represent what was agreed upon during SLA negotiation. They are divided into two categories: Mandatory Terms and Optional Terms. Mandatory Terms specify the terms that are common to all tenants involved in the contract. As for Optional Terms, they encompass the terms that are specific to some tenants. Each term corresponds to a quality attribute and is constituted of one or more Service Level Objectives (SLOs) that specify the quality level expected by the service. Therefore, a mandatory term and an optional term contain, respectively, mandatory SLOs and optional SLOs. An SLO is expressed by means of these attributes: *name*, *threshold*, *unit* and *directionKind*. DirectionKind defines the order relation type; it is an enumeration of *increasing*, *decreasing* and *undefined values* (OMG, 2008). An *increasing* direction means that a value higher than the threshold is the required objective, and vice-versa for the *decreasing* direction. The SLO is constrained by one or more preconditions, which has to be satisfied to ensure the requested quality level.
- Constraints express the dependencies between the different SLOs.

The aim of VariableSLA metamodel is to construct Generic SLAs. Listing 1 represents an example of Generic SLA for Document Management service.

Listing 1: Generic SLA for Document Management service.

```
...
  <parties xsi:type="variableSLA:CloudProvider
      " name="CloudServiceProvider"/>
  <parties xsi:type="variableSLA:Tenant" name
      ="MedicalInsuranceProvider" id="1"/>
  <parties xsi:type="variableSLA:Tenant" name
      ="HealthcareAnalyticsProvider" id="2" />
  <parties xsi:type="variableSLA:Tenant" name
      ="ResearchLaboratory" id="3"/>
  <parties/>
  <services name="DocumentManagementService">
  <terms xsi:type="variableSLA:MandatoryTerm"
      name="Authentication">
    <mandatorySlo name="Login Credentials"/>
  </terms>
       ...
  <terms xsi:type="variableSLA:OptionalTerm"
      name="Performance">
    <optionalSlo threshold="99.95" tenant="
        MedicalInsuranceProvider" name="uptime
        " Unit="%" directionKind="increaisng
        "/>
    <optionalSlo threshold="99" tenant="
        HealthcareAnalyticsProvider" Unit="%"
        directionKind="increasing"/>
    <optionalSlo threshold="98" tenant="
        ResearchLaboratory" name="uptime" Unit
        ="%" directionKind="increasing"/>
  </terms>
  <terms xsi:type="variableSLA:OptionalTerm"
      name="Response Time">
    <optionalSlo threshold="500" tenant="
        MedicalInsuranceProvider" name="
        execution time" Unit="ms"
        directionKind="decreasing"/>
    <optionalSlo threshold="300" tenant="
        HealthcareAnalyticsProvider" name="
        execution time" Unit="ms"
        directionKind="decreasing"/>
    <optionalSlo threshold="800" tenant="
        ResearchLaboratory" name="execution
        time" Unit="ms" directionKind="
        decreasing"/>
  </terms>
  <terms xsi:type="variableSLA:OptionalTerm"
      name="Access Control"/>
  <terms xsi:type="variableSLA:OptionalTerm"
      name="Session Duration Management"/>
  ...
  <constraint expression="Session Duration
      Management requires Access Control "/>
  </services>
</variableSLA:SLA>
```

5.4 NFVariability to VariableSLA Transformation

In order to generate the Generic SLA corresponding to our motivating scenario, we rely on model-to-model transformations performed using ATL language. Model-to-model transformations are based on mapping rules between entities of NFVariability metamodel, the source, and VariableSLA metamodel entities, the target. Table 2 defines these mappings.

Table 2: NFVariability and Variable SLA mappings.

NFVariability Metamodel Entity	VariableSLA Metamodel Entity
Capability	CloudService
Tenant	Tenant
QoSCharacteristic	Term
QoSDimension	SLO
Dependency	Constraint
QoSConstraint	PreCondition

The following VariableSLA elements: Tenant, CloudService and Precondition are copies of their corresponding elements of the NFVariability metamodel. Listing 2 depicts the rule that copies the Tenant *source* to the Tenant *target*.

Listing 2: Tenant2Tenant transformation rule.

```
rule Tenant2Tenant{
       from
       IN: NFVarMM!Tenant
       to
       OUT: VarSLAMM!Tenant(
              id <- IN.tenantId,
              name <- IN.name
       )
}
```

The mapping QoSCharacteristic-Term needs to specify whether the instance of QoSCharacteristic is mandatory or optional so as to generate instances of MandatoryTerm and OptionalTerm. For that purpose, we define two helpers *isMandatory()* and *isOptional()*, mentioned in Listing 3.

Listing 3: isMandatory() and isOptional() helpers.

```
helper context NFVarMM!QoSCharacteristic def:
    isMandatory(): Boolean =
       if self.child.oclIsUndefined() then
              if self.optionality= #mandatory
                  then true
              else false
              endif
       else self.child.including(self) ->
           forAll(q | q.optionality=#mandatory
           )
       endif;
helper context NFVarMM!QoSCharacteristic def:
    isOptional(): Boolean =
       if self.optionality = #optional then
           true
       else false
       endif;
```

Listing 4 presents the transformation rules that transform QoSCharacteristic to MandatoryTerm and OptionalTerm using, respectively, the two aforementioned helpers.

Listing 4: QoSCharacteristic to Term transformation rules.

```
rule QoSCharacteristic2OptionalTerm{
        from
        IN: NFVarMM!QoSCharacteristic (IN.
            isOptional())
        to
        OUT: VarSLAMM!OptionalTerm(
                name <- IN.name
                )
}
rule QoSCharacteristic2MandatoryTerm{
        from
        IN: NFVarMM!QoSCharacteristic(IN.
            isMandatory())
        to
        OUT: VarSLAMM!MandatoryTerm(
                name <- IN.name
                )
}
```

When transforming QoSDimension to SLO, we need to retrieve the threshold values from QoSValue and get the tenants corresponding to those values. Therefore, we define a Tuple type that gathers a quality value with the concerned tenants. Listing 5 describes this transformation.

Listing 5: QoSDimension to SLO transformation rule.

```
helper context NFVarMM!QoSDimension def:
    getValues(): Set(NFVarMM!QoSValue) =
        NFVarMM!QoSValue.allInstances()
       ->collect(v| v.value);
helper context NFVarMM!QoSValue def: getTenant
    (value: String): Set(NFVarMM!QoSValue) =
        NFVarMM!Tenant.allInstances()
       ->select(t | self.value=value);
       helper context NFVarMM!QoSDimension def:
            getTuple(value : String):
           TupleType(v: NFVarMM!QoSValue, t:
           NFVarMM!Tenant) =
               Tuple{v = value,t=value.
                    getTenant()};
rule QoSDimension2SLO{
        from
        IN: NFVarMM!QoSDimension
        to
        OUT: VarSLAMM!SLO(
        name <- IN.name,
        Unit <- IN.unit,
        directionKind <- IN.directionKind,
        threshold <- IN.getValues(),
        tenant <- IN.getValues().first().
            getTuple().t
        )
}
```

The SLA constraints are generated by constructing its expressions from the Dependency relation that may exist between instances of QoSCharacteristic, see Listing 6.

Listing 6: Dependency to Constraint transformation rule.

```
rule Dependency2Constraint{
        from
        IN: NFVarMM!QoSCharacteristic
        to
        OUT:VarSLAMM!Constraint(
        expression <- IN.name + IN.dependency.
            type + IN.dependency.on
        )
}
```

6 CONCLUSION

SaaS applications are generally built on a multi-tenant architecture. This architecture leverages economies of scale but introduces new challenges in adaptation and variability management, especially when it comes to non-functional requirements. In this paper, we proposed an approach that considers Service Level Agreements (SLAs) as Families or Lines. For that, we introduced a novel concept: Generic SLA. We defined two metamodels: NFVariability metamodel and VariableSLA metamodel, to model variability in non-functional requirements and in SLA, respectively. These metamodels form the bases for performing model-to-model transformations from Feature Model to Generic SLA.

Our approach is agile and enables reuse as highlighted through these scenarios that will be tackled in our future work:

- As the Generic SLA forms a product line, it factorizes the commonalities regarding non-functional requirements of tenants, and it specifies the variations among them. This will be exploited to derive tenant-specific SLAs by extracting non-functional properties for a specific tenant.
- In order to cater for dynamic changes in contexts and non-functional requirements of tenants, the SaaS service will be built under context-awareness to detect any changes. If there are any, those changes will be communicated to the Generic SLA so as to incorporate them in the corresponding tenant-specific SLAs after instantiation.
- In case of on-boarding new tenants or appearance of new non-functional requirements of the existing tenants, the Feature Model will be updated to include those new needs and a new Generic SLA is generated.

We will also take into consideration the internal variability, triggered by changes in the infrastructure or cloud provider architectural decisions, so as to define a monitoring process that checks the conformity of the service to the tenant-specific SLAs.

REFERENCES

Abu-Matar, M., Mizouni, R., and Alzahmi, S. (2014). Towards Software Product Lines Based Cloud Architectures. *2014 IEEE International Conference on Cloud Engineering*.

Andrieux, A., CZajkowski, K., Dan, A., and Keahy, K. (2006). Web Services Agreement Specification (WS-Agreement).

Aouzal, K., Hafiddi, H., and Dahchour, M. (2015). An Overview of Variability Management in Cloud Services. *Proceedings of the 2015 International Conference on Cloud Technologies and Applications (CloudTech)*.

Batory, D. (2005). Feature Models, Grammars, and Propositional Formulas. In *Software Product Line Conference*.

Bezemer, C.-P. and Zaidman, A. (2010). Multi-Tenant SaaS Applications : Maintenance Dream or Nightmare ? In *Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE)*.

Boukadi, K., Grati, R., and Ben-Abdallah, H. (2016). Toward the automation of a QoS-driven SLA establishment in the Cloud. *Service Oriented Computing and Applications*.

Fehling, C., Leymann, F., Schumm, D., Konrad, R., and Mietzner, R. (2011). Flexible process-based applications in hybrid clouds. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*.

Galster, M. (2015). Architecting for Variability in Quality Attributes of Software Systems. *Proceedings of the 2015 European Conference on Software Architecture Workshops*.

Galster, M., Weyns, D., Tofan, D., Michalik, B., and Avgeriou, P. (2014). Variability in Software Systems: A Systematic Literature Review. *Software Engineering, IEEE Transactions on Software Engineering*.

García-Galán, J., Pasquale, L., Trinidad, P., and Ruiz-Cortés, A. (2014). User-centric Adaptation of Multi-tenant Services: Preference-based Analysis for Service Reconfiguration. *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*.

García-galán, J., Pasquale, L., Trinidad, P., and Ruiz-Cortés, A. (2016). User-Centric Adaptation Analysis of Multi-Tenant Services. *ACM Transactions on Autonomous and Adaptive Systems*.

Gey, F., Van Landuyt, D., Walraven, S., and 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 colocated with 17th International Conference on Model Driven Engineering Languages and Systems MODELS*, Valencia, Spain.

Horcas, J. M., Pinto, M., and Fuentes, L. (2016). An automatic process for weaving functional quality attributes using a software product line approach. *Journal of Systems and Software*.

Horcas, J.-M., Pinto, M., and Fuentes, L. (2017). Green Configurations of Functional Quality Attributes. *Proceedings of the 21st International Systems and Software Product Line Conference - Volume A on - SPLC '17*.

Keller, A. and Ludwig, H. (2003). The WSLA Framework : Specifying and Monitoring Service Level Agreements for Web Services.

Landuyt, D. V., Walraven, S., and Joosen, W. (2015). Variability Middleware for Multi-tenant SaaS Applications. *Proceedings of the 19th International Systems and Software Product Line Conference - SPLC '15*.

Mahdavi-Hezavehi, S., Galster, M., and Avgeriou, P. (2013). Variability in quality attributes of service-based software systems: A systematic literature review. *Information and Software Technology*.

Mell, P. and Grance, T. (2011). The NIST Definition of Cloud Computing Recommendations of the National Institute of Standards and Technology.

Mohamed, M., Anya, O., Tata, S., Mandagere, N., Baracaldo, N., and Ludwig, H. (2017). rSLA: An Approach for Managing Service Level Agreements in Cloud Environments. *International Journal of Cooperative Information Systems*.

OMG (2008). Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms.

Pohl, K., Böckle, G., and van der Linden, F. (2005). *Software Product Line Engineering: Fundations, Principles and Techniques*. SpringerVerlag, Berlin, DE.

Serrano, D., Bouchenak, S., Kouki, Y., De Oliveira, F. A., Ledoux, T., Lejeune, J., Sopena, J., Arantes, L., and Sens, P. (2016). SLA guarantees for cloud services. *Future Generation Computer Systems*.

Tata, S., Mohamed, M., Sakairi, T., Mandagere, N., Anya, O., and Ludwiga, H. (2016). RSLA: A service level agreement language for cloud services. *IEEE International Conference on Cloud Computing, CLOUD*.

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

Uriarte, R. B., Tiezzi, F., and Nicola, R. D. (2014). SLAC: A Formal Service-Level-Agreement Language for Cloud Computing. In *2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing (UCC)*.