Towards Modeling a Variable Architecture for Multi-Tenant SaaS-Applications

Julia Schroeter, Sebastian Cech, Sebastian Götz, Claas Wilke and Uwe Aßmann
Institute for Software- and Multimedia-Technology
Technische Universität Dresden
D-01062, Dresden, Germany
{julia.schroeter, sebastian.cech, claas.wilke, uwe.assmann}@tu-dresden.de,
sebastian.goetz@acm.org

ABSTRACT

A widespread business model in cloud computing is to offer software as a service (SaaS) over the Internet. Such applications are often multi-tenant aware, which means that multiple tenants share hardware and software resources of the same application instance. However, SaaS stakeholders have different or even contradictious requirements and interests: For a user, the application's quality and non-functional properties have to be maximized (e.g., choosing the fastest available algorithm for a computation at runtime). In contrast, a resource or application provider is interested in minimizing the operating costs while maximizing his profit. Finally, tenants are interested in offering a customized functionality to their users. To identify an optimal compromise for all these objectives, multiple levels of variability have to be supported by reference architectures for multi-tenant SaaS applications. In this paper, we identify requirements for such a runtime architecture addressing the individual interests of all involved stakeholders. Furthermore, we show how our existing architecture for dynamically adaptive applications can be extended for the development and operation of multi-tenant applications.

Keywords

Multi-Tenancy, Software-as-a-Service, Variability Modeling, Auto-Tuning, Self-Optimization

1. MOTIVATION

Delivering Software as a Service (SaaS) is a widespread business model in cloud computing [4, 17]. One special type of SaaS applications are single-instance multi-tenant applications (MTAs). In a single-instance MTA, the same software and hardware resources are shared between multiple tenants [6]. Tenants are interest groups or companies that rent a tailored application from the SaaS application provider. A tenant provides the tailored application to his own group of potential users (e.g., for in-house usage or to vend the application to customers). For the individual users of a certain tenant, service and resource sharing is transparent (i.e., they are unaware of the fact that other users—even from other tenants or other applications—may use the same resources and applications in parallel).

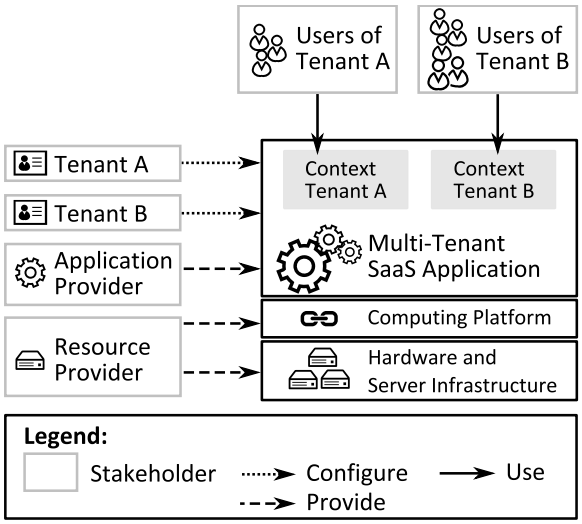

Figure 1: Stakeholders and their Activities in a Multi-Tenant SaaS Application.

In the context of MTA, different types of stakeholders can be identified (cf. Figure 1): *Resource providers* offer server infrastructure, including computing platforms (e.g., operating systems, databases and component containers) as platform services. *Application providers* develop and deploy the SaaS application on top of them. *Tenants* rent the application. Due to multiple tenants' functional and non-functional requirements may differ, such an application must be configurable. The respective configurations are represented at runtime by tenant contexts (i.e., concrete functionality that is available in the tailored application). These contexts comply with the constraints defined for their tenants including the selection of optional functionality (software components) and business-related constraints like the usage of European servers only. Finally, *users* belonging to a certain tenant access the application without the need of any installation on their own machines. For tenant- and user-specific customization they use the application in an enclosing tenant

context as well as an open individual user context. Such user contexts cover non-functional requirements of their users like a minimum framerate for video playback or a maximum response time for certain operations.

For the different stakeholders of an MTA different objectives can be identified: From a user's point of view, the application qualities have to be maximized (e.g., choosing the fastest available algorithm for a computation). Different tenants are interested in customizing available services to their individual needs (e.g., they enable or disable additional services or tailor the application's design to their own corporate design) [27]. From a resource provider's point of view, operational costs of the infrastructure should be minimized (e.g., using less energy or minimizing third-party license fees that are payed per used CPU). Finally, application providers are interested in maximizing the number of tenants and their users while minimizing the utilized resources in order to maximize their profit (e.g., by sharing as much components as possible). It should be clear that at least some of these requirements are contradictious (e.g., providing optimal performance whereby reducing operation costs) and have to be negotiated between the individual stakeholders.

To allow this negotiation process, MTAs require variability to be expressed on different levels of abstraction in the problem space and in the solution space. In the problem space, variability in functionality need to be modeled, as tenants have different functional requirements. Therefore, well established variability modeling techniques, like feature modeling [14, 3], decision modeling [22, 23], and orthogonal variability modeling (OVM) [16] are convenient on this level of abstraction. Furthermore, in the solution space, variability among component implementations on the architectural level need to be expressed [19]. Even though component implementations usually serve the same functional requirements, they differ in their non-functional properties (NFPs) and hardware requirements at runtime. The problem we tackle in this paper is, how to realize a variable and configurable reference architecture for MTAs, that adapts automatically at runtime to different tenant configurations as well as their users' requirements. Thereby, the architecture needs to preserve their contexts and satisfies all MTA stakeholders.

In this paper we identify requirements for MTA architectures and propose a reference architecture to address them. We show that several of these requirements can be addressed by state-of-the-art frameworks for dynamically adaptive systems (i.e., software systems that automatically adapt themselves to modified requirements or usage contexts at runtime) [10, 12, 20] and thus, can be reused and extended for MTA architectures. We use multi-quality auto-tuning (MQuAT) as one example for such a framework and propose extensions of its architecture to address identified but yet unsupported requirements for MTAs.

The remainder of this paper is structured as follows: We give an illustrative example for an MTA in Sect. 2. Following in Sect. 3, we identify general requirements for MTA-aware architectures. Afterwards, we present our quality auto-tuning architecture in Sect. 3 and discuss required extensions to support MTA. We present related approaches for MTA architectures and discuss the advantages of MQuAT in contrast to other frameworks for dynamically adaptive systems in Sect. 4. Finally, we conclude our work in Sect. 5.

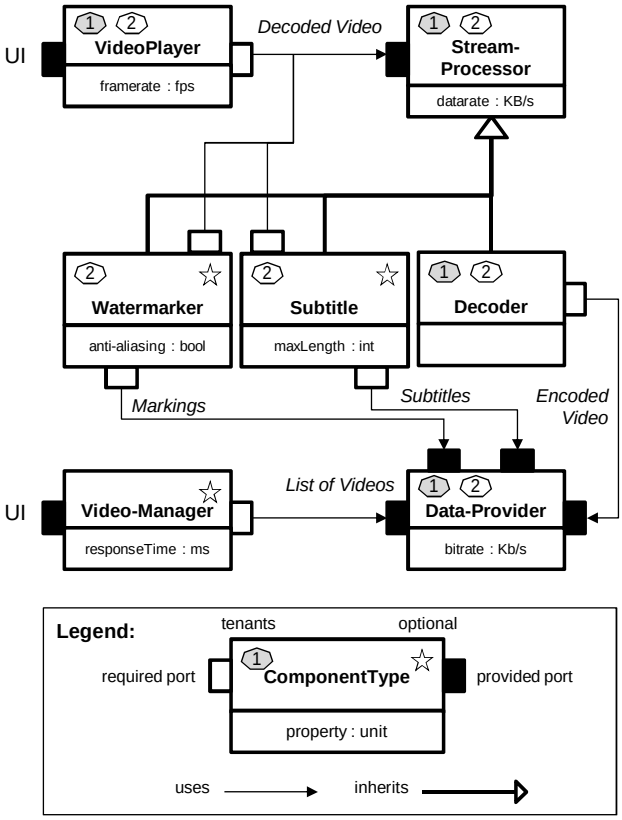

Figure 2: VideoPortal Multi-Tenant Application Including two Tenant Configurations.

2. ILLUSTRATIVE EXAMPLE

To exemplify our approach we introduce a video-portal MTA in this section, which will be used throughout the paper. In our scenario, an application provider offers a configurable video portal application to his tenants. The application is comprised of several components as depicted in Figure 2. The components will shortly be explained in the following. The presented component types are emphasized, their implementations are formatted in italics font, and properties in capitalized font.

Tenants can choose different subsets of the components, whereof all have to include a *Player*, *Decoder* and *Data-Provider* component, which form the core of the video application. Additional components (demarcated by a star in Figure 2) include a *VideoManager* component, which offers a graphical user interface to add and remove videos or to change their properties, a *Watermark* component, which injects a tenant-specific watermark into the videos before they are played, and a *Subtitle* component, which introduces subtitles as overlays on videos.

In addition, a *StreamProcessor* component is required, of which the *Decoder*, *Watermarker* and *Subtitle* components inherit. This abstraction enables the tenant to choose a combination of the three inheriting components. All components inheriting from *StreamProcessor* provide a decoded video, but only the *Decoder* component decodes an encoded video an thus, does not require a decoded video by itself. Hence, all combinations of the three inheriting components

Table 1: Implementation Variants of the Component Types in the VideoPortal Multi-Tenant Application.

Component Type	Implementations
VideoPlayer	VLC, WMP
Decoder	Free, Commercial
DataProvider	URL, File
Watermarker	TransparentMarker, ClassicMarker
Subtitle	SingleLanguage, MultiLanguage
VideoManager	Standard, Professional

are valid, if a *Decoder* component is included. To avoid endless chains of *Watermarker* and *Subtitle* components, multiplicity constraints can be used, which are annotated at each component type.[1] Notably, every component potentially has multiple implementations. This is due to the fact that all implementations of the same component type provide the same functionality, but differ in their non-functional properties. For example, the *VideoPlayer* component is realized by a *VLC* and a *WMP* implementation. Both implementations play a video, but *VLC* offers a different video resolution than *WMP* does. Further implementations are listed in Table 1.

For each tenant an own configuration of the application can be created. For our example, we assume two tenant configurations, depicted in Figure 2 by the numbered heptagons in each component: the first tenant (T1) does not use any optional component, whereas the second tenant (T2) has decided to use the *Watermarker* and *Subtitle* component to enhance the application.

Besides provided and required functionality (via ports), each component defines a set of NFPs (c.f. Figure 2), which characterize how the provided functionality can be fulfilled. For example, the *VideoPlayer* component defines a property FRAMERATE, which characterizes how fast the videos are played (i.e., how many frames per second can be shown). The *Watermarker* component has a Boolean property ANTI-ALIASING, expressing whether tenant-specific markings can be smoothed when being integrated into the original video stream using anti-aliasing.

3. VARIABLE ARCHITECTURE

In this section, we identify requirements for variable multi-tenant aware architectures. Eventually, we describe an adaptive architecture that meets these requirements accordingly. As the context of our identified requirements, we only consider MTAs under the following conditions [1, 6, 7]:

- **Cloud Infrastructure** We assume a cloud infrastructure, i.e., a server cluster, whereat server nodes are distributed over various locations and are able to communicate with each other. If a server fails, affected applications are automatically switched to another node.
- **Scalable Server Cluster** The server cluster supports horizontal scaling, i.e., server nodes are added to the cluster or removed, according to their utilization.
- **Measurable NFPs** We assume, that we are able to measure the NFPs of all resources in the cloud at any time.
- **Single-Instance Application** To realize multi-tenancy, we suppose a configurable single-instance strategy on application level, whereas the application instance is shared by multiple tenants, but its components allow for tenant-specific configuration and customization.
- **Distributed Application** An MTA instance is distributed over multiple physical server nodes in the server cluster.
- **Scalable Application at Runtime** At design time of an MTA, concrete tenant configurations that will be contained in the single-instance application are unknown. Whereas, at application runtime, new tenant configurations are created and added to the application instance.

[1] Multiplicities are not shown in Figure 2 for clarity.

3.1 Requirements on the Architecture

To address the variability identified in Sect. 1 and based on the assumptions described above, we identify the following requirements that a component-based MTA architecture has to fulfill. We distinguish between requirements at design time and requirements at runtime. At design time an MTA has to be described by using an appropriate modeling language whereas at runtime models are used to optimize the different stakeholders' objectives introduced in Sect. 1.

3.1.1 Design Time Requirements

/DR1/ **Software and Hardware Component Modeling** A modeling language needs to provide concepts to describe software and hardware components. Software component modeling is performed by the application provider to describe the architecture of the MTA. Tenants may have business-related constraints regarding the underlying hardware infrastructure on which the software components are executed. Therefore the resource provider has to create an infrastructure model. Furthermore, this infrastructure model can be used to address the objectives of the resource provider.

/DR2/ **Specification of NFPs** Next to functional requirements, tenants and their users have non-functional requirements on software and hardware components. This requires that an MTA modeling language must be able to specify NFPs for software and hardware components as well as dependencies between them. Software components may depend on other software components as well as on hardware components, whereas hardware components may depend on other hardware components only.

/DR3/ **Tenant Configuration Modeling** Tenant configurations have to be modeled at design time, too. A tenant configuration describes the tenant by a unique identifier, the architectural software components that are available for this tenant and constraints regarding NFPs at software component and hardware component level. The explicit modeling of tenant configurations is needed to preserve this information and to manage multiple tenants and their users at runtime.

3.1.2 Runtime Requirements

/RR1/ **Self-Optimizing Runtime Environment** A variable architecture for MTA applications requires a self-optimizing runtime environment that is able to reason about the current hardware infrastructure, the application architecture and tenant contexts. Furthermore, the architecture must be able to optimize itself regarding the distribution of software components depending on current tenant contexts. In addition, the runtime environment must be scalable to multiple tenants and their users. It must be capable of handling multiple user requests at the same time.

/RR2/ **Monitoring of NFPs** In order to reason on software and hardware components and tenant contexts, NFPs specified at design time have to be updated periodically. Hence, values of NFPs have to be monitored at runtime and propagated through the runtime environment.

/RR3/ **Management of Tenants and Users** The architecture has to support multiple tenants and multiple users at the same time. Thus, the architecture must be able to manage tenant as well as user contexts. That is the basis for balancing the objectives of multiple tenants and users.

/RR4/ **Multi-Tenant Specific Platform Services** At runtime, an MTA architecture requires several services like authorization, authentication or a multi-tenant capable database [7]. As these services are used by all tenants, they should be available as platform services.

/RR5/ **Resource Sharing** Multiple hardware and software components of the same MTA instance are shared between tenants. The tenants' users should not influence each others resource utilization.

3.2 Multi-Quality Auto-Tuning Architecture

The MQuAT architecture, former referred to as energy auto-tuning (EAT) architecture [12], provides a good basis for our envisioned MTA architecture. Its main constituents are the cool component model (CCM), the quality contract language (QCL) (former referred to as energy contract language (ECL)) and the three layer auto tuning runtime environment (THEATRE). This framework for dynamic adaptive systems was initially developed with the focus on energy, but has been extended to handle multiple qualities.

We decided to use MQuAT and extend it to fit requirements of MTA applications due to the following reasons. First, it is a component based auto-tuning (AT) architecture (initially introduced in [13]) realizing self-optimization of the components' NFPs. Second, it explicitly considers hardware in addition to software components using contracts, which characterize dependencies between the systems components in terms of non-functional requirements. Third, it offers adaptivity at deploy and runtime, as will be shown in the following.

3.2.1 Component Model and Component Contracts

The CCM is a meta-model covering various aspects for quality-efficient, self-optimizing software-systems. For clarity, we focus on structure and variant models as defined in the CCM in this paper. In addition, we discuss the QCL as a contract language, which is used to define dependencies between the system's components. Figure 3 depicts a simplified overview of the MQuAT architecture's component model.

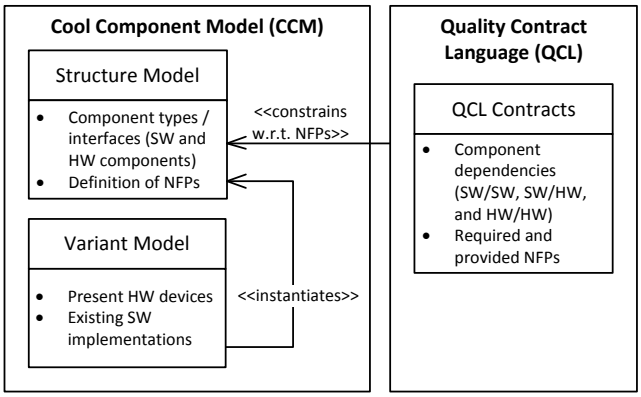

Figure 3: Simplified overview of the CCM and the QCL concepts.

To specify the types of components a system can comprise, structure models of the CCM exist (e.g., to specify the `DataProvider` component type of the video portal example depicted in Figure 2). Structure models are used to specify hardware component types, too. Such models usually contain a `Server` component type and may contain further types like `Router` or `TabletPC`. Notably, component types are hierarchical organized. Thus, the description of a `Server` comprises descriptions for a `CPU` and `HDD` to name but a few. Furthermore, each component type defines NFPs as exemplified in Sect. 2.

A concrete variant of the system is described by variant models. They comprise component implementations or resources (i.e., instances of component types) and refer to the respective component types in a structural model. For each property defined in the component type, the instance offers a value depending on the kind of property. The CCM distinguishes three types of properties: static-instance, monitored and calculated. If a property has an immutable value for each instance (e.g., the maximum frequency of a CPU) it is called a static-instance property. If the value needs to be monitored at runtime (e.g., the current frequency of the CPU) it is called a monitored property. Finally, if the value of a property can be computed using other properties it is called a calculated property. Furthermore, a component instance contains the information where/if it is currently deployed.

Although structure and variant models allow describing the current state of the complete system, they do not suffice to reason about the system's optimality. This is, because we need knowledge about how the components depend on each other. This knowledge can be specified using QCL contracts. Each contract comprises a set of quality modes, which are modes of perception by the requesting component (or user). Listing 1 shows an example contract for the *VLC* video player. The FRAMERATE of a video can be perceived as fluent, hesitant or slideshow. Each of these modes is characterized by a different minimum provided FRAMERATE with different requirements on properties of other compo-

```
1  VideoPlayer { mode fluent {
2      //required resources
3      requires resource CPU {
4        max cpuLoad = 50 percent
5        min frequency = 2 GHz
6      }
7      requires resource Net {
8        min bandwidth = 10 MBit/s
9      }
10     //dependencies on other SW components
11     requires component Decoder {
12       min dataRate = 50 KB/s
13     }
14     //what is provided in turn
15     provides min frameRate 25 Frame/s
16   }
17   mode hesitent { ... }
18   mode slideshow { ... }
19 }
```

Listing 1: Example Contract for VLC Video Player.

nent types. That is, each mode defines property provisions of its own as well as property requirements of other software and hardware components.

Thus, the CCM and QCL fulfill the design time requirements /DR1/ and /DR2/, as they allow for the modeling of hardware and software (CCM) as well as for the specification of NFPs (CCM) and the dependencies between components (QCL). Requirement /DR3/ is not supported by CCM and QCL, yet. The changes to the architecture needed to support this requirement will be discussed in Sect. 3.3.

3.2.2 Runtime Environment

THEATRE is a runtime environment for self-optimizing software systems. That is, it realizes a control loop with four constituting phases—collect, analyze, decide and act—as described by Dobson et al. in [8]. First, information about the system has to be collected, which is analyzed subsequently. Based on these analyzes the system decides how to proceed by deferring a plan and realizing it (act). To realize the collect phase, THEATRE offers hierarchically organized resource managers which use profiling techniques to collect data about the available resources. The analysis phase is realized by evaluating QCL contracts. Decisions are deferred by contract negotiation [11]. In our terms, contract negotiation is used to derive the optimal system configuration (i.e., a selection of implementations and their mapping to resources) for a specified user request and his demands. It is realized as a constraint solving optimization problem (CSOP), which is generated based on the information provided by the structure and variant models as well as the involved contracts. Finally, the act phase is realized by component-based reconfiguration (component migration) and a role-based adaptation technique, which we are currently investigating.

The runtime environment comprises three layers: a user, a software, and a resource (i.e., hardware) layer as depicted in Figure 4. The software and resource layers contain a set of hierarchically organized managers. The resource layer is managed by *resource managers*, whereof the topmost manager in the hierarchy is called the global resource manager. These managers provide the information which resources exist as well as their characteristics. Additionally, they provide means to control these resources (e.g., shutting down

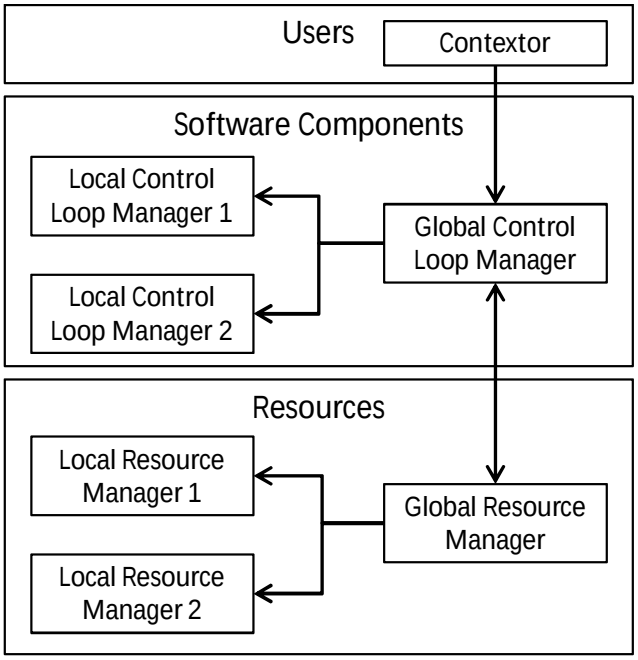

Figure 4: The Layers of THEATRE and their Respective Managers.

a server or manually or scaling a CPU's frequency). The software layer is managed by *control loop managers*. The topmost manager is called the global control loop manager, which is connected to the global resource manager. Control loop managers know about all available software component types and implementations, the mapping of implementations to servers/component containers and are responsible for the realization of the control loop. That is, they initiate and manage the loop by (1) collecting information using the global resource manager, (2) analyzing the collected data, (3) determining the optimal system variant (using contract negotiation) as well as (4) deriving and executing the respective reconfiguration plan.

The user layer covers feature requests and associated user demands in terms of non-functional requirements. This information is collected by *contextors*. Notably, the current version supports single user operation only. Techniques for modeling the interplay of concurrent usage by multiple users and the respective manifestation by interleaving NFPs on the level of soft- and hardware components are tasks of current work. In consequence, multi-tenancy is not supported, yet, too.

Thus, THEATRE offers support for the runtime requirements /RR1/, which is realized by the control loop manager hierarchy and /RR2/, which is realized by the resource manager hierarchy. The requirements /RR3/, /RR4/ and /RR5/ are not supported, yet.

In conclusion, the MQuAT architecture as described in detail in [12] fulfills four of eight requirements[2] of our envisioned MTA architecture. In the next section we will describe potential extensions to realize the remaining requirements.

[2]Two of three design time requirements and two of five runtime requirements are fulfilled.

3.3 Extending the Architecture

As we have shown in the previous section, the MQuAT architecture already addresses multiple requirements of an MTA architecture. However, it neither supports the modeling of tenant configurations (/DR3/) nor the management of tenants (/RR3/). In addition, it does not yet provide tenant-specific platform services (/RR4/) nor the sharing of resources (/RR5/). To address these requirements, we propose an extension which effects the current architecture at three locations. To allow for the modeling of tenant configurations, the concepts for structure models in the CCM are extended with multi-tenant concepts. Handling tenants at runtime requires an extension to the runtime environment THEATRE and to the concepts for variant models in the CCM in accordance to the changes to the structure model concepts. This is because variant models refer to structure models like models refer to their meta-model (or instances to their type). Finally, tenant-specific services [7, 15] like authentication, are realized as components, which have to be present in each MTA. Thus, they are mandatory MTA components. This allows for different implementations of these components and their integration into the auto-tuning approach (i.e., an optimal implementation for the particular user context is used at runtime).

Notably, the mere modeling of tenant configurations does not suffice to realize a self-reconfigurable SaaS application. The provided knowledge has to be considered throughout the autonomous control loop [8]. Especially the *decide* phase has to be adjusted for MTAs. This is, because at runtime the optimizer has to consider the constraints defined by the tenant (e.g., his components must not run on a server outside of Germany). This implies the need for a constraint language, which empowers the tenant to specify his constraints. In addition, the optimizer has to consider shared use of software components by tenants and the resulting implications w.r.t. the provision of NFPs. There is no need to adjust the other three phases (*collect*, *analyze* and *act*), as tenants are neither monitored nor effected at runtime (only their users).

In the following we will first show the required extensions to the CCM and discuss required extensions to THEATRE. Next, we will elaborate on the domain-specific constraint language for tenant configuration constraints. We do not discuss the realization of tenant-specific services like authentication, because state-of-the-art components can be used for that purpose.

3.3.1 Extending CCM

To identify missing concepts for multi-tenancy in CCM, a close investigation of the tenant configuration concept is used as a starting point.

A *tenant configuration* describes the functionality that is available to a tenant and its users and constraints the executing environment of the MTA. In other words, it limits the variant space of CCM. A tenant configuration tc is defined as a 4-tuple in the form that $tc = \langle id_t, c_{arch}, c_{env}, d_{app} \rangle$. id_t is a unique tenant id, c_{arch} is a set of constraints on selectable software component types and their implementation variants. c_{env} is a set of constraints on measurable NFPs of the software and hardware component types and finally, d_{app} is a reference to the concrete MTA, where the configuration will be integrated. A tenant configuration is instantiated at runtime as a tenant context.

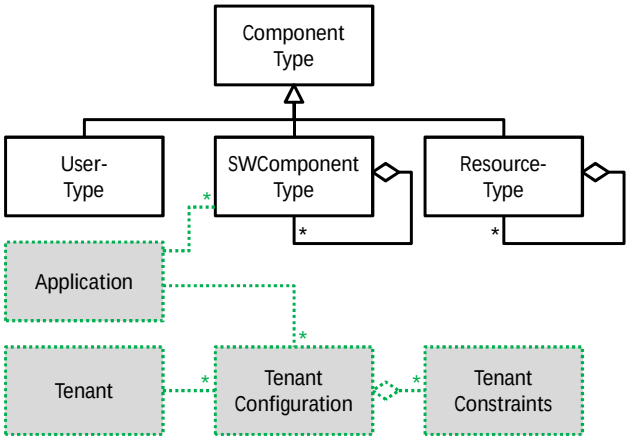

Figure 5: Extensions to CCM Structure Modeling for MTA Applications.

Extensions to Structure Modeling.

Figure 5 depicts the extended structure modeling part of the CCM (extensions are highligthed with dotted lines). To model tc, the metaclass `TenantConfiguration` is introduced, which tailors an MTA for a certain `Tenant` (representing id_t). In turn, multiple `TenantConfiguration`s may exist for each `Tenant`. The MTA is represented by the metaclass `Application` (d_{app}), which may refer to multiple optionally or obligatory available `SWComponentTypes`. It is tailored in multiple `TenantConfiguration`s. A `TenantConfiguration` consists of multiple `TenantConstraints`, in order to express the functional (c_{arch}) and non-functional (c_{env}) constraints. Both are expressed using the proposed constraint language and are presented in Sect. 3.3.2.

Regarding the example introduced in Sect. 2, the video portal MTA is modeled as an `Application`. Component types of this MTA including its ports and connections between ports are captured in a CCM structure model as `SWComponentType`s. Notably, mandatory and optional components are already modeled in the problem space, i.e. they are not focussed here. Each tenant (T1 and T2) of this application is described by an instance of the corresponding metaclass containing exactly one `TenantConfiguration`. Tenant T2 has the architectural constraint that the `Watermarker` and the `Subtitle` component should be included in its configuration. Furthermore, assume that T2 has environmental requirements, e.g. the whole application has to be run on servers located in the European Union (EU) with high availability end strong encryption support. Such requirements are modeled in terms of `TenantConstraint`s.

Extensions to Variant Modeling.

To manage the system at runtime, the concepts of the variant modeling part need to be extended, too (cf. Figure 6). Extensions are highlighted using dotted lines. Imports from the structure model and QCL are marked by dashed lines.

Variant models express the runtime state of the system and need to be able to express, which tenant configurations are currently active (i.e., in use). Thus, the metaclasses `TenantContext` and `UserContext` are introduced. A `TenantContext` represents the runtime instance of a `TenantConfiguration` and refers to its current `User`s, currently

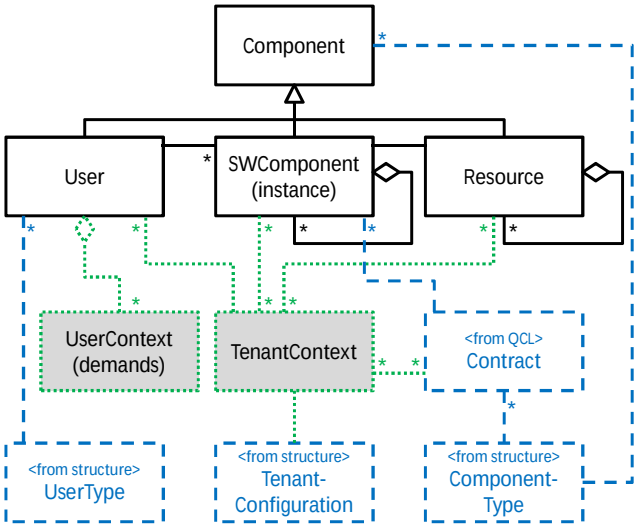

Figure 6: Extensions to CCM Variant Modeling for MTA Applications.

used `SWComponents` and `Resources`. This allows expressing knowledge about which users are currently using which `SWComponents` on which `Resources`. Each `User` has multiple `UserContexts` expressing which functionality of the MTA is being requested by which user as well as his respective non-functional requirements. The evaluation of the tenant constraints—that are included in a tenant configuration—at runtime, results in a subset of all existing resources and software component contracts, which represent available implementations. These subsets are used for contract negotiation (instead of the complete sets) as will be explained in Sect. 3.3.3. The subsets are represented by the associations between `TenantContext` and `Contract` as well as `Resource`.

Considering the example from Sect. 2, a valid `TenantContext` for tenant T2 contains only software components deployed on highly available servers located in the EU with encryption facilities. In contrast, a `TenantContext` for tenant T1 may contain only core components of the video portal, whereas the server location is not constrained. For users of the MTA variability is still available and need to be bound according to their request. For instance, according to the required FRAMERATE (e.g. 25 fps) specified in a user's request, the implementation of the *Player* component type, that offers at least this framerate will be choosen in the `UserContext`.

The presented extensions address /DR3/ and allow covering almost all required information to handle MTA applications by THEATRE. The remaining aspects of MTA applications are covered by tenant configuration constraints.

3.3.2 Constraint Language

Functional as well as non-functional requirements of a tenant need to be expressed by constraints in a *constraint language*. The constraints define software component types and potential implementations that are available for a tenant's users. In addition, non-functional requirements, e.g., availability, performance and security, are defined as well. A common type of constraint is the ex- or inclusion of servers based on their location, due to legal rules (e.g., only servers located in the EU). Availability is a special kind of constraint, too. It is considered as a constraint, because it restricts the number of possible system configurations as it requires replication. Tenant-specific security rules are another kind of constraint. Data isolation is a general security requirement for SaaS applications, but the degree of encryption used can differ per tenant. That is, a tenant could request strong encryption either at the cost of performance or by paying for utilizing more resources. All constraints, both functional and non-functional, are expressed using propositional formulas [2]. Referring to the example introduced in Sect. 2, tenant T2 has architectural c_{arch} and environmental c_{env} constraints, that are specified in separate formulas. From an architectural point of view, all component types—except the $VideoManager$—are available in T2's tenant configuration. This is expressed using the formula $c_{arch} = VideoPlayer \wedge Decoder \wedge DataProvider \wedge Watermarker \wedge Subtitle$. It has to be noted that the variability is constrained at software component type level. However, it may be constrained at implementation level, too. For instance, if only one implementation (e.g., $SingleLanguage$) of the subtitle component is available for a tenant is expressed using the notation $Subtitle.SingleLanguage$. In the example, the environmental constraints of T2 specify that only servers located in the EU should be used. Other NFPs values of tenant T2 are high availability and strong encryption. Those constraints are expressed using the formula $c_{env} = available.high \wedge encryption.strong \wedge server.location.eu$.

To evaluate a constraint, the implications of each proposition are used to filter all possible system configurations. For instance, all servers whose location property is not equal to "eu" are excluded. The implications of high availability refer to the minimum cardinality of deployed implementations (replication), strong encryption implies the existence of encryption components on each involved server and a long security key as a parameter of the encryption component.

3.3.3 Extending THEATRE

THEATRE utilizes the extensions to the CCM and the tenant constraints as they provide the required additional information w.r.t. tenant configurations. The runtime environment needs to be adjusted, so it evaluates the tenant constraints at runtime and thereby limits the search domain of the contract negotiation by prefiltering resources, software implementations and computing minimum cardinalities as well as additionally required components.

The reduced number of resources and software implementations leads to less linear constraints in the CSOP realizing contract negotiation. In turn, minimum cardinalities and additionally required components lead to supplemental linear constraint. Thus, the CSOP generator, as a part of the control loop manager, has to be extended to include the prefiltering by tenant constraint evaluation.

The presented extensions address the runtime requirements /RR3/ and /RR4/. Notably, resource sharing by users of tenants (/RR5/) can be considered a constraint, too. Thus, to realize /RR5/, a standard tenant constraint can be used. This implies the assignment of as much users to component instances as possible (while considering the user's minimum non-functional requirements). We currently examine the implications on component instances w.r.t. their non-functional properties in the context of multiple, concurrent users.

3.3.4 Discussion

The realization of tenant constraints by prefiltering available resources and component implementations is only one possible solution. Another approach is the incorporation of constraints as NFPs into the models. This can be done by introducing NFPs for encryption, availability and location. The user request, which includes the user's demands, is implicitly enriched with the tenant's demands. That is the `TenantContext` has to comprise non-functional requirements as is the case for the `UserContext`. A user request in a certain tenant context leads to a user context, which is enriched with these non-functional requirements.

The consequence of this alternative approach are additional constraints in the CSOP, expressing which resources and component implementations must not be used. But in contrast to the prefiltering approach, the solution/search space of the CSOP is not reduced. Therefore, we suggest to introduce a constraint language, whose constraints are used to prefilter the search space of the CSOP.

4. RELATED WORK

An application architecture for MTAs is described in [18]. In that approach a service component architecture (SCA) is extended with variability descriptors and multi-tenancy patterns to package and deploy configurable applications that are multi-tenant aware, as composite applications. In contrast, our component-based architecture is adaptable to different tenant contexts as well as to different user requests at runtime.

Another service-based architecture for MTAs is proposed in [26]. The authors use a hypergraph-based service model to represent services and MTAs. They distinguish functional and business dependencies between services. In addition, our approach defines component contracts based on NFPs.

Tsai et al. [25] describe an architecture for cloud computing. They propose that a single multi-tenant application instance uses multiple distributed service instances on different clouds. Their tenant configurations define, which service instances they should use. In contrast, in our approach the service instances are defined variable within the tenant configuration. We simply define that a certain component type is required. The concrete instances are determined on demand according to the required NFPs.

An approach to monitor a multi-tenant single-instance application as well as resource allocation based on service level agreements (SLAs) is proposed in [5]. The provided architecture consists of a performance monitor, a component to detect abnormal states of the system and a scheduling component to schedule tenants' user access on shared resources. The tenants are prioritized w.r.t. tenants' SLA. Users of tenants with high priority will gain earlier access to shared resources. In our apporach resources are shared, too. But if the resource utilization is too high, a new instance of that component will be instantiated due to the auto-tuning capability of our architecture. Furthermore, we consider not only performance, but multiple qualities.

Managing the life cycle of service-based applications is explained in [21]. Ruz et al. provide a framework for flexible service management of business processes on the cloud including the design and the monitoring of applications and their components. They use a monitor to measure NFPs of the components and present them to the configuration manager. If a non-functional requirement is not fulfilled, this manager triggers a reconfiguration manually. In our approach, we use contract negotiation to automate this process.

The DiVA research project resulted in a framework for dynamic adaptive systems by combining methods from aspect-oriented programming and model-driven software development (MDSD) [20]. DiVA allows for automatic adaptation of systems at runtime, supporting goal-based optimization as well as rule-based reconfiguration of the system w.r.t. NFPs [9]. An extension of DiVA for MTAs similar to our approach would be possible. However, DiVA covers symbolic NFPs only (e.g., low, medium, and high framerates) whereas our approach supports the optimization of sub-symbolic (i.e., numeric) NFPs instead (e.g., 22fps instead of a *high* or *medium* framerate). This pays off—especially for MTAs—where multiple stakeholders with different interests are involved as their interests can be negotiated more fine granularly using sub-symbolic optimization.

Another framework for dynamic adaptive systems emerged from the MUSIC research project. It provides a component model for self-adaptive applications on mobile devices [10] which can be used to describe multiple component variants and their distribution on available devices in accordance to the NFPs. MUSIC systems are optimized at runtime to provide maximum user satisfaction (but do not negotiate satisfaction with implied costs). Nevertheless, w.r.t. the identified requirements for an MTA architecture, MUSIC fulfills the same of the identified requirements as our existing MQuAT architecture. But, in contrast to MUSIC, the MQuAT architecture allows for user-defined types of devices (i.e., servers, mobile devices, humanoid robots to name but a few) and, thus, provides better extensibility.

5. CONCLUSION AND FUTURE WORK

In this paper we presented our vision of a variable and adaptive SaaS architecture. We identified requirements that a multi-tenant aware SaaS reference architecture must support at design time as well as at runtime. At design time certain application information needs to be modeled, e.g., an application's software and hardware components and their properties as well as tenant configurations that define which of these components will be available for their users and how they are constrained. Runtime requirements include the possible evaluation of the modeled information to manage the tenants and their users as well as to self-optimize the runtime environment according to a tenant's configuration and his users' demands. Therefore, the NFPs of the components must be monitored. A main requirement of an MTA at runtime is the support for sharing resources among multiple tenants and their users. Still, there is the need for multi-tenant specific platform services, e.g., to authorize a user and assign him to the correct tenant configuration.

As our research shows, the component-based MQuAT architecture is already able to address several of these requirements. Hence, we outlined multi-tenant specific extensions of the architecture to cover all requirements. We are currently implementing the proposed extensions. We are also working on case studies to evaluate the applicability of our approach. In addition, we will apply it on large scale scenarios involving various tenants and users to test its scalability. An open research question is, how tenant-specific customizations can be preserved on the evolution of an MTA.

In future work, we will manage the variability among an MTA on a higher level of abstraction. From a software product line (SPL) point of view, the MTA's tenant configurations constitute a product family. Thus, the combination of this architecture with SPL techniques is promising to automate the configuration process. We will use a feature-oriented generative approach to automate the process of deriving tenant configurations almost automatically as motivated in [24]. Because SPL engineering is a well researched field, we may benefit from developed tools that help to derive valid tenant configurations.

Our goal in the long term is to ease the handling of MTAs by supporting their development, configuration and maintenance with appropriate tools. We consider the variable architecture proposed in this paper is a good basis for this challenge.

6. ACKNOWLEDGEMENTS

The authors would like to thank the reviewers for their valuable comments and suggestions. The work presented in this paper is part of the industry-supported doctoral project #080949335, and the projects ZESSY #080951806, CoolSoftware #FKZ13N10782, and the DFG-funded Collaborative Research Center 912 (HAEC). The project #080949335 is co-funded by the European Social Fund, Federal State of Saxony and SAP AG. The ZESSY project is funded by the European Social Fund and Federal State of Saxony. The project CoolSoftware is part of the Leading-Edge Cluster "Cool Silicon", which is sponsored by the Federal Ministry of Education and Research (BMBF) within the scope of its Leading-Edge Cluster Competition.

7. REFERENCES

[1] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. Above the clouds: A berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, 2009.

[2] D. Batory. Feature models, grammars, and propositional formulas. In *Proceedings of the International Software Product Line Conference*, SPLC '05, pages 7–20. Springer, 2005.

[3] D. Benavides, S. Segura, and A. Ruiz-Cortés. Automated analysis of feature models 20 years later: A literature review. *Information Systems*, 35(6):615–636, 2010.

[4] K. Bennett, P. Layzell, D. Budgen, P. Brereton, L. Macaulay, and M. Munro. Service-based software: the future for flexible software. In *Proceedings of the Asia-Pacific Software Engineering Conference*, APSEC '00, pages 214–221, 2000.

[5] X. Cheng, Y. Shi, and Q. Li. A multi-tenant oriented performance monitoring, detecting and scheduling architecture based on SLA. In *Proceedings of the Joint Conferences on Pervasive Computing*, JCPC '09, pages 599–604, 2009.

[6] F. Chong and G. Carraro. Architecture strategies for catching the long tail. MSDN Website, 2006. Retrieved September 15, 2011, from `http://msdn.microsoft.com/en-us/library/aa479069.aspx`.

[7] F. Chong and G. C. nd Roger Wolter. Multi-tenant data architecture. MSDN Website, 2006. Retrieved September 15, 2011, from `http://msdn.microsoft.com/en-us/library/aa479086.aspx`.

[8] S. Dobson, S. Denazis, A. Fernández, D. Gaïti, E. Gelenbe, F. Massacci, P. Nixon, F. Saffre, N. Schmidt, and F. Zambonelli. A survey of autonomic communications. *ACM Transactions on Autonomous and Adaptive Systems*, 1:223–259, 2006.

[9] F. Fleurey and A. Solberg. A domain specific modeling language supporting specification, simulation and execution of dynamic adaptive systems. In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems*, MODELS '09, pages 606–621. Springer, 2009.

[10] K. Geihs, M. U. Khan, R. Reichle, A. Solberg, S. Hallsteinsen, and S. Merral. Modeling of component-based adaptive distributed applications. In *Proceedings of the ACM Symposium on Applied Computing*, SAC '06, pages 718–722. ACM, 2006.

[11] S. Götz, C. Wilke, S. Cech, and U. Aßmann. Runtime variability management for energy-efficient software by contract negotiation. In *Proceedings of the International Workshop on Models@run.time*, MRT '11, 2011. to appear.

[12] S. Götz, C. Wilke, S. Cech, and U. Assmann. *Sustainable Green Computing: Practices, Methodologies and Technologies*, chapter Architecture and Mechanisms for Energy Auto Tuning. IGI Global, 2011. to appear.

[13] S. Götz, C. Wilke, M. Schmidt, S. Cech, and U. Aßmann. Towards energy auto tuning. In *Proceedings of the International Conference on Green Information Technology*, GREEN IT '10, 2010.

[14] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report, Carnegie Mellon University Pittsburgh, Software Engineering Institute, 1990.

[15] H. Koziolek. Towards an architectural style for multi-tenant software applications. In *Software Engineering'10*, pages 81–92, 2010.

[16] K. Lauenroth and K. Pohl. *Software product line engineering - foundations, principles, and techniques*, chapter 4, pages 57–86. Springer, 2005.

[17] D. Ma. The business model of "software-as-a-service". In *Proceedings of the IEEE International Conference on Services Computing*, SCC '07, pages 701–702. IEEE Computer Society, 2007.

[18] R. Mietzner, F. Leymann, and M. P. Papazoglou. Defining composite configurable saas application packages using sca, variability descriptors and multi-tenancy patterns. In *Proceedings of the International Conference on Internet and Web Applications and Services*, ICIW '08, pages 156–161, 2008.

[19] R. Mietzner, A. Metzger, F. Leymann, and K. Pohl. Variability modeling to support customization and deployment of multi-tenant-aware software as a service applications. In *Proceedings of the Workshop on Principles of Engineering Service Oriented Systems*, PESOS '09, pages 18–25, 2009.

[20] B. Morin, O. Barais, G. Nain, and J.-M. Jézéquel. Taming dynamically adaptive systems using models and aspects. In *Proceedings of the International Conference on Software Engineering*, ICSE '09, pages 122–132. IEEE Computer Society, 2009.

[21] C. Ruz, F. Baude, B. Sauvan, A. Mos, and A. Boulze. Flexible soa lifecycle on the cloud using sca. In *Proceedings of the IEEE International Enterprise Distributed Object Computing Conference Workshops*, EDOCW '11, pages 275–282, 2011.

[22] K. Schmid and I. John. A customizable approach to full lifecycle variability management. *Science of Computer Programming - Special issue: Software variability management*, 53:259–284, 2004.

[23] K. Schmid, R. Rabiser, and P. Grünbacher. A comparison of decision modeling approaches in product lines. In *Proceedings of the Workshop on Variability Modeling of Software-Intensive Systems*, VaMoS '11, pages 119–126. ACM, 2011.

[24] J. Schroeter. Towards generating multi-tenant applications. In *Pre-Proceedings of the GTTSE/SLE Students' Workshop*, GTTSE/SLE '11, 2011.

[25] W.-T. Tsai, X. Sun, and J. Balasooriya. Service-oriented cloud computing architecture. In *Proceedings of the International Conference on Information Technology: New Generations*, ITNG '10, pages 684–689. IEEE Computer Society, 2010.

[26] R. Wang, Y. Zhang, S. Liu, L. Wu, and X. Meng. A Dependency-Aware Hierarchical Service Model for SaaS and Cloud Services. In *Proceedings of the IEEE International Conference on Services Computing*, SCC '11, pages 480–487, 2011.

[27] B. Waters. Software as a service: A look at the customer benefits. *Journal of Digital Asset Management*, 1:32–39, 2005.