Composite as a Service: Cloud Application Structures, Provisioning, and Management

Composite as a Service: Strukturen, Provisionierung und Management von Cloud Anwendungen

Christoph Fehling, University of Stuttgart,
Ralph Retter, T-Systems International GmbH, Leinfelden-Echterdingen

Summary Cloud computing and corresponding "as a service" models have transformed the way in which IT resources can be consumed. By taking advantage of the properties of the cloud – elasticity, pay-per-use and standardization – customers and providers alike can benefit from economies of scale, faster provisioning times and reduced costs. However, to fully exploit the potentials of the cloud, it is necessary, that applications, to be deployed on the cloud, support the inherent cloud properties. In this article we investigate how applications can be designed to comply with cloud infrastructures. We present a framework that allows modeling the variability within such applications regarding their structure, functional, and non-functional properties, as well as their deployment. Using these models the framework guides the user during the customization of an application, provisions it on available clouds, and enables common management functionality for cloud applications, such as elasticity, suspend, and resume. ►►► **Zusammenfassung** Cloud Computing und die dazugehörigen Geschäftsmodelle haben die Art und Weise, in der IT Ressourcen genutzt werden, dramatisch verändert. Aufgrund der Cloud spezifischen Eigenschaften, wie Elastizität, flexiblen Preismodellen und Standardisierung, können Anbieter und Kunden gleichermaßen von Skaleneffekten, kürzeren Bereitstellungszeiten und Kostenreduktion profitieren. Um allerdings diese positiven Eigenschaften von Clouds ausnutzen zu können, ist es notwendig, dass diese auch innerhalb der Anwendung berücksichtigt werden. In diesem Artikel stellen wir ein Framework vor, mit dem die Variabilität solcher Anwendungen bezüglich ihrer Struktur, funktionalen und nicht-funktionalen Eigenschaften, sowie ihres Deployments modelliert werden kann. Auf Basis der hierzu erstellten Modelle begleitet das Framework den Nutzer während der Anpassung der Anwendung an seine Bedürfnisse und provisioniert sie auf verfügbaren Clouds. Weiterhin nutzt es die Modelle, um kundenspezifische Managementfunktionalität, wie Elastizität, Suspend und Resume, zu realisieren.

Keywords C.0 [Computer Systems Organization: General]; C.2.4 [Computer Systems Organization: Computer-Communication Networks: Distributed Systems]; D.2.2 [Software: Software Engineering: Design Tools and Techniques]; D.2.3 [Software: Software Engineering: Coding Tools and Techniques]; D.2.7 [Software: Software Engineering: Distribution, Maintenance, and Enhancement]
►►► **Schlagwörter** Cloud Computing, Strukturen, funktionale Eigenschaften, nicht-funktionale Eigenschaften, Verwaltungsarchitektur

1 Introduction

With the advent of cloud computing the vision of accessing computing as a utility gathered widespread acceptance. The fundamental properties of cloud-based infrastructures and platforms include elasticity – the ability to dynamically scale based on customer demand, pay-per-use – the fact that customers pay only for resources they consume, and standardization – the advent

of standardized infrastructure, platform, software artifacts, and management interfaces that enable providers to exploit economies of scale. Depending on the user group that has access to a certain cloud one refers to them as being private – organization internal, public – available to everyone, or hybrid – a combination of private and public cloud resources. Due to these different realizations, clouds differ greatly in non-functional properties such as price, performance, and privacy.

The services provided by elastic cloud infrastructures are generally categorized as infrastructure, platform, and software as a service depending on the portion of the application stack that is controlled by the cloud provider. Infrastructure as a Service (IaaS) offerings, such as Amazon EC2, Rightscale or GoGrid, as well as tooling to build such infrastructures, like Eucalyptus, Zimory, and IBM Tivoli Service Automation Manager, form the basis of many cloud-based applications today. The same holds true for Platform as a Service (PaaS) offerings, such as Microsoft Azure, Google AppEngine or Saleforce's Force.com platform. Such infrastructure and platform offerings are paramount and corresponding optimization and deployment techniques are widely discussed. There is less discussion on the influences that these elastic infrastructures and platforms have on the structures, the provisioning, the management, and the execution of applications running on top of such offerings. In particular, there is little discussion on how applications have to be designed so that they can benefit from the special properties of clouds and the corresponding service models. For example, elasticity of applications depending on infrastructure as a service is mainly realized on the infrastructure level rather than on the application level.

Given the availability of IaaS, PaaS and SaaS today, applications composed of components built on multiple of these models start to emerge. If such applications are again offered to customers in an as a service model, it becomes mandatory that the application is customizable regarding its functional and non-functional properties to reach a large number of consumers. Due to the different non-functional properties of clouds, this customization requires a customer specific deployment of the application. For different customers different clouds and different levels of sharing components among tenants are acceptable. Therefore, a customer specific composition of application components and service models is required. Also, possible enhancements of the application structure might be required that ensure elasticity or handle workload management, for example.

In this paper we discuss applications built of components that use different infrastructures and platforms and show how to compose them into a composite application that is again offered as a service (CaaS – Composite as a Service). The service level of this new service may be IaaS, PaaS, or SaaS depending on the portion of the application stack that has been composed. We motivate why such applications are needed and their benefits and drawbacks. Further it is shown how such applications can be modeled and annotated with variability to be customizable regarding their functional and non-functional properties. We present our corresponding models for application topologies and variability that we developed for Cafe (composite application framework), a framework to model, configure and automatically provision composite applications. To cope with cloud-specific properties such as elasticity, we introduce extensions to the Cafe framework in this paper that allow to semi-automatically derive additional management workflows. These management workflows handle cloud-specific properties of CaaS applications such as the provisioning, suspending/resuming, reconfiguration, growing, shrinking or de-provisioning of individual application components or whole applications.

2 Related Work

Infrastructure as a service is offered by companies such as Amazon (EC2, S3), GoGrid and Rightscale. Several research projects have dealt with the effective management [11; 12] and optimization and deployment [5] of IaaS resources.

In the PaaS field, the major players include Google with its AppEngine, Salesforce with its Force.com platform, Microsoft with its Azure platform and Amazon with some of its Web Services (SQS, RDS). Research has been done on how to build platforms for the deployment of multi-tenant enabled applications [2; 14] which is also the focus for the research in the SaaS field.

Players in the SaaS field include Salesforce with its CRM product, Google with its Google Apps, IBM with Lotus live, Microsoft with Office Live, and Deutsche Telekom with its DeveloperGarden. Composite applications have been researched extensively in the SOA domain [6; 13]. These applications mainly consist of services that are recursively aggregated into new composite services, i. e., through orchestration languages such as WS-BPEL. However, these approaches do not take the specifics of the "as a service" models into account as they effectively only aggregate services on the software layer. With the advent of IaaS and PaaS, composite applications can be aggregated not only out of services (in the SaaS model) but also out of infrastructure and platform services. Such composite applications can then again be offered as a service (CaaS) [6; 10]. Application and deployment models, such as the one presented in [3; 5], can be used to model such applications but most of the time do not take multi-tenancy into account. In the Cafe project [7; 10] we introduced models that allow to model composite applications that make use of different IaaS, PaaS and SaaS services. We also showed, similar to [3], how required initial provisioning actions can be derived from the models. However, these approaches do not describe how to derive management actions from the application model.

In autonomic computing significant research has been performed to enable self-* properties of distributed sys-

tems [4]. Entities in such systems undergo a MAPE-loop (monitor, analyze, plan, execute) to adjust their behavior and their direct surrounding to optimize the overall system regarding its scalability, availability, performance etc. Optimization of the overall system therefore emerges from the distributed optimization decisions of the entities it is composed of. These approaches assume a homogeneous runtime environment, but composite applications often have to integrate resources from versatile computing environments. For example, an application could be provisioned mainly on a private cloud but during peak loads also integrates resources of a public cloud. Under such conditions enabling of self-* properties based on local knowledge of application components is extremely difficult [1]. The approach discussed in this paper therefore considers a centralized management approach that also supports the distribution of management functionality among distributed components when their local knowledge of the system is sufficient and enacts central control otherwise.

3 Application and Variability Metamodels for CaaS

In this section we briefly introduce the application and variability metamodels of the Cafe project [7; 10]. The *application metamodel* allows to model the application topology for a composite application, i. e., out of which components the application consists and which component has to be deployed on which other component. The *variability metamodel* allows to define the variability of the application. We then show how these models can be extended to support cloud-specific management functionality, such as growing or shrinking (elasticity).

3.1 Application Modeling

In [10] we describe how the structure and deployment topology of applications can be modeled using the Cafe application model. First of all the application developer has to model the *components* of the application. This includes application components such as UIs, Workflows, Services, Databases, Message Queues as well as their *implementations* in form of code files or other artifacts. Middleware and hardware components such as servers, database management systems, application servers, workflow engines or messaging systems have to be modeled additionally. Components can have special *implementations* of *implementation type provider-supplied* which indicates, that the provider has to provision this component and that the code for the component is not supplied within the application. Components with an *implementation type* of *provider supplied* have to specify a *component type*. The *component type* tells the provider what kind of component he has to supply. The *multi-tenancy pattern* indicates whether a component can be shared with other customers or not. *Deployment relations*, i. e., which components can be deployed on which other component, also have to be modeled in the application model. This allows

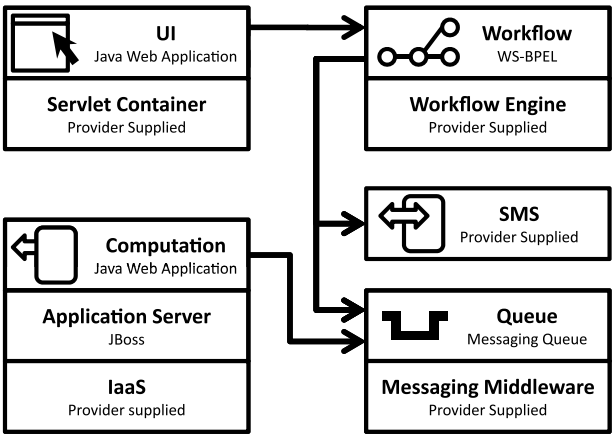

Figure 1 Architecture of the example application.

modeling, for example, that a UI component has to be deployed on an application server. Components C and the deployment relations $D \subseteq C \times C$ in an application model form a directed, acyclic deployment graph $DG = (C, D)$.

Figure 1 shows a fictional example CaaS application. It is offered to customers in a self-service portal where they can subscribe and unsubscribe from this application that is then provisioned or de-provisioned on demand. The application consists of a *UI* component that is implemented as a *Java Web Application* and must be deployed on a *Servlet container* that supports such applications such as the Google AppEngine or a Tomcat servlet container. The UI component forwards requests to a *workflow* component that must be deployed on a corresponding workflow engine and orchestrates two Web services. The first service, realized by the *computation* component is also realized as a Java Web application and must be deployed on a corresponding *AppServer* component, such as a JBOSS application server running in an IaaS offering such as Amazon EC2. As the *computation* service is performing computing intensive operations and thus has to scale elastically with the number of requests, it is separated from the *workflow* by a *queue* component, realizing a message queue. This component must be deployed on a *messaging middleware* such as Amazon's SQS. The second service orchestrated by the *workflow* component is an *SMS* Web Service that notifies a user of the completion of a workflow instance. It is offered in the *SaaS* model, for example by DeveloperGarden. The *UI, workflow, queue* and *computation* components contain their respective implementation in the application model. The other components are of implementation type *provider supplied* which means that the provider has to supply a corresponding implementation or delegate this to third party (cloud) providers.

3.2 Variability Modeling

Another challenge in defining CaaS applications is to define their variability. To capture the variability offered in such applications it is important to understand the different variability requirements that arise from the inherently

distributed nature and heterogeneity of such applications. We tackle four classes of variability: *functional variability*, *non-functional variability*, *deployment variability* and *structural variability*.

- Functional variability is variability that impacts the program logic or presentation of a program, for example the addition of steps to the *workflow* component in our example, or the modification of the *UI* component.
- Non-functional variability is the variability that affects the service level of an application such as availability, performance etc.
- Deployment variability is variability that must be bound during the provisioning and management of the application, such as the binding of endpoint references. In our example application, the *computation* component as well as the *workflow* component must be configured with the address of the *queue* component. In this case the deployment variability affects the order in which components must be provisioned, as first the *queue* component must be provisioned so that its address is known so that it can be used to configure the two other components.
- Structural variability is variability that affects the application model of a component. For example, a customer could replace the *SMS* component with another communication component that is running in his private cloud.

The different classes of variability can affect each other. For example, a functional variability can add additional steps to the *workflow* component which mandates an additional *service* component to be deployed which is structural variability as it would add the component to the application model. Additionally, it triggers the binding of deployment variability as the endpoint reference of the new component must be configured in the workflow component. We therefore advocate the use of an *orthogonal variability metamodel* that can capture all four classes, across different components and different providers. We introduced such a metamodel in the Cafe project in [7; 9; 10]. According to this metamodel a variability model for a CaaS application consists of a set of *variability points VP*. Each *variability point* is associated with a *component* from the corresponding application model. For each *variability point* the *phase* it must be bound in is annotated. Allowed phases are: *initial customization* of the application by a customer, *pre-provisioning* of the particular component by the management infrastructure, *post-provisioning* of the particular component by the management infrastructure and *deprovisioning* of the particular component by the management infrastructure. *Variability points* can point into the implementation artifacts of a component or can be used by a provider to determine how he has to provision a certain component. Each *variability point* contains a set of *alternatives* that represent choices a customer can make. *Free alternatives* allow customers to enter arbitrary values. *Dependencies* $Dp \subseteq VP \times VP$ allow to describe dependencies between *variability points*, i. e., which *variability point* must be bound before which other *variability point*. *Variability points* and their *dependencies* form a directed, acyclic *variability graph* $VG = (VP, Dp)$. *Enabling conditions* allow to refine these *dependencies* by activating certain *alternatives* only when a certain *condition* is met. Thus, one can specify conditions such as "if alternative $A1$ has been selected at variability point A, you can only select alternative $B2$ at variability point B".

3.3 Extensions to the Cafe Application Model

In this section we describe novel extensions to the Cafe application model we briefly described above and more extensively described in [7; 10]. These extensions are necessary to cope with the dynamic nature of CaaS applications in the cloud, namely the necessity to add or remove components dynamically during runtime, based on certain *triggers*. Triggers can be *user-initiated*, i. e., a user wants to suspend/resume an application, or wants to scale-up or down an application. The user-initiated initial provisioning of an application can be seen as another type of user-initiated trigger. Triggers can also be *time-initiated*, i. e., a component or a whole application is suspended during the night and resumed in the morning. Additionally triggers can be *system-initiated*, i. e., a certain event, such as a degradation in performance, or the reaching of a message queue's threshold size fires the trigger. We allow to model such *triggers* in the application model. For each trigger we allow to model a complex *condition* that evaluates if one or more *notifications* have been received that initiate the trigger. Such notifications can be sent by users through the management portal or can be sent by components. For example, the *queue* component can be annotated with *notifications* that state that a lower or upper bound of messages is reached in the queue and that the corresponding *queueOverload* or *queueEmpty* trigger should be fired.

For each component in the application model we can define *actions* that describe what should happen if a certain *trigger* fires. For example, we can specify that an additional *computation* component has to be deployed on an *AppServer* component that has to be provisioned once the *queueOverload* trigger fires. We will describe the concrete operations that can be executed by a component in the next section.

4 Management of CaaS Applications

In this section we introduce a management architecture for CaaS applications and show how the application and variability models introduced in Sect. 3 can be exploited by corresponding tooling to support the different phases of the lifecycle of a CaaS application.

4.1 Management Infrastructure

A centralized management infrastructure at a provider depicted in Fig. 2 offers a self-service portal to customers,

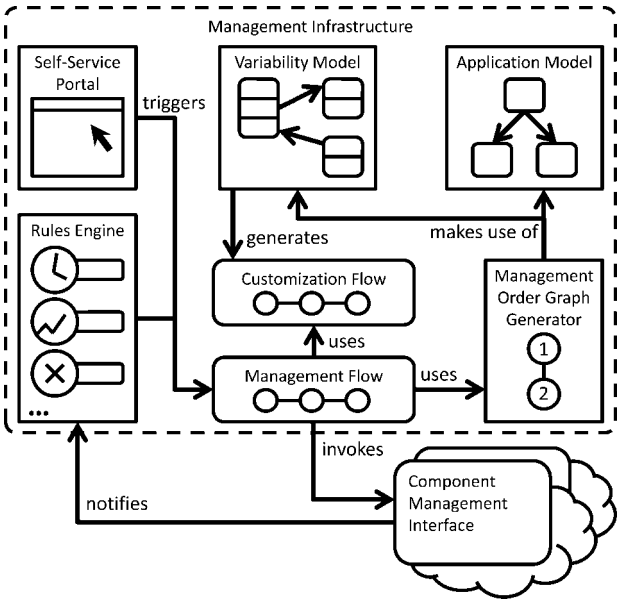

Figure 2 Architecture of the management infrastructure.

from which they can manage their applications. In particular, they can configure and provision new applications from the portal. Providers can offer different CaaS applications in the portal that are modeled according to the models described in Sect. 3. The management infrastructure contains *management flows* that execute *management requests* sent to them by the *portal* or the *rules engine*. The *rules engine* is essentially a stream processor that sends *management requests* to the management flows if a certain *condition* fires. These conditions are taken from the *triggers* in the application model. The *management request* then contains information on which trigger fired. Similar to that *management requests* from the portal contain information which user-initiated trigger fired.

Two types of *management flows* can be distinguished. *Application-vendor specific management flows* and the *default management flow*. Application-vendor specific management flows are supplied by the application vendor and can contain complex management actions. The default management flow is supplied by the portal and can execute management actions that require only the knowledge that is contained in the application and variability models of an application. We describe default management flows and the *management order graph generator* on which they depend in detail in Sect. 4.2.

Management flows can make use of *customization flows* to collect customization values that are needed to configure components. Customization flows are specific for each application and are generated from the variability model of an application. We describe how customization flows are generated in more detail in [8].

Management flows also make use of *component management interfaces* for the components that are affected by a management request. The provider has to offer a *component management interface* for each *component type* he can provision. This component management interface must offer standardized operations to *reserve*, *provision*, *suspend*, *resume*, and *deprovision* components as well as to *get configurations* and *configure* a component. Components that allow that other components are deployed on them must additionally offer a *deploy* and *undeploy* operation. Components that are multi-tenant aware must offer *addTenant* and *removeTenant* operations. The component management interface abstracts from the concrete management tooling used to manage a component which can be monitoring tools, provisioning engines or the API of a cloud provider such as Amazon's EC2 API. For more information on the concrete *component management interfaces* see [7; 8]. We implemented wrappers implementing this interface for a variety of management interfaces such as a Amazon EC2, a cloud managed by Zimory, Eucalyptus and Sun N1 Service Provisioning System.

4.2 Default Management Flow

In our previous research for the Cafe project we found out that a variety of applications can be provisioned and deprovisioned with the knowledge included in the application model and the variability model of the application [7; 10]. Thus we were able to use a default provisioning flow to provision such applications. Here we show how the knowledge included in the application model and the variability model can be used to cope with a variety of additional cloud-related management requests and thus can be handled by a default management flow generalizing the concept of the default provisioning flow.

On a high level, the default management flow works as follows: (i) invoke the customization flow to get the current customization of the application, (ii) invoke the management order graph generator to get the components that are affected and the order in which they must be treated and then (iii) perform the management actions for each component.

As each management request contains information about the trigger that initiated it, the management flow first needs to know which components are affected by the trigger. Therefore it forwards the trigger to the *management order graph generator (MOGG)*. The MOGG then inspects for each component in the application model if it specifies an *action* for that trigger. If yes it adds the component and its action to the *affected component list*. There exist four implicit triggers *initialProvisioning, deProvisioning, suspend* and *resume*. In these cases all components are added to the *affected component list*, either with the operation contained in the corresponding action (if specified) or with the operation that corresponds to the management request. For example, for an *initialProvisioning* request, all components that do not have an action that is triggered by that trigger are added with an operation of *provision*. In the second step, the MOGG determines the order in which components need to be treated.

The order in which components need to be treated is determined by the application model and the variability model. The general rule is that a component that depends on other components must be provisioned or deployed after the components it depends on, for example, if it must be deployed on another component as specified in the application model. The other case of dependencies is if a component is dependent on another one because of the variability model. It then has a variability point that must be bound before the component must be provisioned but after the component it depends on has already been provisioned. In case components must be deprovisioned or suspended, first those must be handled that depend on other components and then those can be treated when all components that depend on them have already been deprovisioned.

In case some components must be provisioned for a management request and some must be deprovisioned, first all components are provisioned in the right order, and then the other ones are deprovisioned.

In the following we describe some concrete management requests for our example application, and how they are handled by the default management flow and the MOGG:

Initial Provisioning. During initial provisioning the customization flow has never been invoked before, thus it asks the customer for all necessary decisions. Then the MOGG returns a management order graph that contains all components. Each component that needs to be deployed on another component (all those with an implementation contained in the application model) is annotated with the operation "deploy", those that need to be provisioned (those with an implementation type of *provider supplied*) are annotated with the operation "provision". If the application would contain components shared with other tenants and these were already available, those would be only marked with "configure" indicating that they must be configured, rather than provisioned or deployed. The corresponding implementations of the operations then have to take care of the deployment and provisioning of the respective components. In case of the *queue* component that contains *notifications* these have to be used to configure the corresponding management framework to send out the notifications to the management infrastructure when the queue reaches the lower or upper thresholds.

Suspend. The MOGG returns a list of all components in the order they must be suspended (reverse order of initial provisioning) with an annotated action of "suspend". All those components shared with other tenants are excluded.

Elasticity. A notification is received by the rules engine that the *queue* component has reached the upper threshold. The corresponding rule fires and sends the *queueOverload* trigger to the management flow. The management flow then forwards this trigger to the MOGG to receive a list of affected components and required actions, namely the *computation* component and its underlying *AppServer* component. It then has to provision the *AppServer* component first and then configure the *computation* component with the right queue address and deploy it on the *AppServer* component. The reverse actions are performed when the *queueEmpty* trigger is sent.

Once the order in which components have to be treated is returned by the MOGG, the management flow interprets the returned management order graph and executes the necessary actions in the right order. In case components must be configured, the management flow invokes the customization flow to ask for the values necessary to customize the component. These customization values are then attached to the invocation of the component management interface to supply the underlying management tools with the necessary configuration data. These management tools then perform the corresponding operations, for example, provision a new component. When the operation is finished, the management flow is notified. Configuration values, such as, for example, the endpoint reference of a newly provisioned component, are returned to the management flow which forwards them to the customization flow. This is necessary as other variability points from other components might depend on these values and these components can then be configured. Once all components in the management order graph have been handled and thus all necessary management operations have been invoked in the right order, the management request that triggered the management flow has been fulfilled.

In our prototype the management flow is implemented as a BPEL process that runs on an Apache Ode BPEL engine and makes use of a Java Web Service that implements the provisioning order graph generator. Customization flows are implemented as BPEL processes that are generated from the variability model of an application. The self-service portal is implemented as a Java Web application.

5 Summary

We have seen how application components that use the different service models of clouds as a runtime can be composed to form new applications that again can be offered as a service. These applications have been designed with customizability in mind, thus their variability has been modeled using the application and variability models of the Cafe framework. Using these models the framework deducted a customization flow that guides the customer of the application through the configuration of this variability using a self-service portal to reflect his individual requirements. Further, an extension to the application model was introduced to model triggers that perform certain actions in case of certain conditions in the environment or their initiation by the user or at a certain time. Since these triggers were also targeted by

application customization, individualized management behavior could be realized by the framework to support the automatic provisioning and deprovisioning of the application as well common management functionality for cloud applications such as elasticity, suspend, and resume.

References

[1] K. Begnum, N. Lartey, and L. Xing. Cloud-Oriented Virtual Machine Management with MLN. In: *Cloud Computing*, pages 266–277, 2009.

[2] F. Chong and G. Carraro. Architecture strategies for catching the long tail. *MSDN Library, Microsoft Corporation*, 2006.

[3] K. El Maghraoui, A. Meghranjani, T. Eilam, M. Kalantar, and A. V. Konstantinou. Model driven provisioning: Bridging the gap between declarative object models and procedural provisioning tools. In: *Proc. of the ACM/IFIP/USENIX Int'l Conf. on Middleware*, pages 404–423, 2006.

[4] J. O. Kephart and D. M. Chess. The vision of autonomic computing. In: *Computer*, 36, pages 41–50, 2003.

[5] A. V. Konstantinou, T. Eilam, M. Kalantar, A. A. Totok, W. Arnold, and E. Snible. An architecture for virtual solution composition and deployment in infrastructure clouds. In: *Proc. of the 3rd Int'l Workshop on Virtualization Technologies in Distributed Computing (VTDC)*, pages 9–18, 2009.

[6] F. Leymann. Cloud Computing: The Next Revolution in IT. In: *Proc. of the 52th Photogrammetric Week*, pages 1–10. Online, 2009.

[7] R. Mietzner. *A method and implementation to define and provision variable composite applications, and its usage in cloud computing*. Dissertation, University of Stuttgart, August 2010.

[8] R. Mietzner and F. Leymann. Generation of BPEL Customization Processes for SaaS Applications from Variability Descriptors. In: *Proc. of the 2008 IEEE Int'l Conf. on Services Computing (SCC)*, pages 359–366, 2008.

[9] R. Mietzner and F. Leymann. Towards Provisioning the Cloud: On the Usage of Multi-Granularity Flows and Services to Realize a Unified Provisioning Infrastructure for SaaS Applications. In: *Proc. of the IEEE Congress on Services*, pages 3–10, 2008.

[10] R. Mietzner and F. Leymann. A Self-Service Portal for Service-Based Applications. In: *Proc. of the IEEE Int'l Conf. on Service-Oriented Computing and Applications (SOCA)*, pages 1–8, 2010.

[11] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov. The eucalyptus open-source cloud-computing system. In: *Proc of the 9th IEEE/ACM Int'l Symp. on Cluster Computing and the Grid*, volume 0, pages 124–131, 2009.

[12] B. Rochwerger, D. Breitgand, E. Levy, A. Galis, K. Nagin, I. M. Llorente, R. Montero, Y. Wolfsthal, E. Elmroth, J. Calceres, M. Ben-Yehuda, W. Emmerich, and F, Galan. The RESERVOIR Model and Architecture for Open Federated Cloud Computing. In: *IBM Journal of Research and Development*, 53(4):535–545, 2009.

[13] F. Rosenberg, P. Leitner, A. Michlmayr, P. Celikovic, and S. Dustdar. Towards Composition as a Service – A Quality of Service Driven Approach. In: *Proc. of the 25th Int'l Conf. on Data Engineering (ICDE)*, pages 1733–1740, 2009.

[14] W. Sun, X. Zhang, C. J. Guo, P. Sun, and H. Su. Software as a service: Configuration and customization perspectives. In: *Proc. of the IEEE Congress on Services*, pages 18–25, 2008.

Received: November 23, 2010

Dipl.-Inf. Christoph Fehling graduated from the University of Stuttgart in 2009 with a Diploma in Computer Science. He currently researches cloud application architectures and the runtime management of cloud applications at the Institute of Architecture of Applications Systems (IAAS) at the University of Stuttgart.

Address: University of Stuttgart, Institute of Architecture of Application Systems, Universitäts straße 38, 70569 Stuttgart, Germany, Tel.: +49 711 685-88486, Fax: +49 711 685-88472,
e-mail: christoph.fehling@iaas.uni-stuttgart.de

Dr. Ralph Retter geb. Mietzner is an architect at T-Systems working on cloud applications. Before joining T-Systems Ralph was a researcher at the Institute of Architecture of Application Systems (IAAS) at the University of Stuttgart. His research was centered around cloud applications, their structures and the automatic configuration and provisioning of such applications. Ralph holds a Dr. rer. nat. from University of Stuttgart.

Address: T-Systems International GmbH, Fasanenweg 5, 70771 Leinfelden-Echterdingen, Germany, Tel.: +49-711-972 46756,
e-mail: ralph.mietzner@t-systems.com