A Self-Service Portal for Service-Based Applications

Ralph Mietzner, Frank Leymann
Institute of Architecture of Application Systems
University of Stuttgart
70563 Stuttgart, Germany
{*mietzner, leymann*}*@iaas.uni-stuttgart.de*

***Abstract*—Enterprises today constantly seek to reduce the cost of their IT-operations. One popular approach is to outsource (parts) of these IT-operations. With the advent of Cloud computing, the outsourcing of infrastructure, runtime platforms and even whole applications has been greatly facilitated. With platforms such as Amazon's EC2 or Salesforce's Force.com customers can select from a set of pre-defined machine images or applications that they can then run on-demand. However, all these platforms employ proprietary technology that does not permit to model, offer, configure and automatically deploy applications in the cloud in a *generic* manner. In this paper we introduce metamodels, algorithms and tools for application vendors to describe composite service-based cloud applications independently of the provider that later offers them. We describe how customers can customize such service-based applications and how providers can automatically provision the individual services required by the application, thus offering customized applications to their customers while minimizing costs by sharing services and infrastructure across customers. We report on the architecture and implementation of our prototype and show sample applications.**

Keywords-Cloud computing; composite service-based applications; application portals; provisioning; customization

I. Introduction

Cloud computing [1] is often characterized by offering *elasticity*, i.e. the ability to scale up and down *on demand* as well as *as a service models* where the initial expenditures for a service as well as operation and management of that service are delegated to the provider. The goal is that customers only pay for what they really use (*pay-per-use model*). These three properties, elasticity, as a service and pay-per use have driven cloud success stories such as Amazon EC2 (on the infrastructure level), Google App Engine (on the platform level) or Salesforce (on the software level). As enterprises seek to reduce their capital expenditures (CAPEX) in favor of operational expenditures (OPEX), they try to outsource standard applications (such as CRM) to third party providers. However, as different enterprises have different requirements and different business processes, these standard applications must be customizable to suit the needs of the enterprises [2] and to integrate them with services already present in the enterprise. At the same time providers seek to maximize their profits by industrializing their operations. By offering a standard set of services for a large customer base, providers can exploit economies of scale. In this paper we combine the concepts of application portals as they can be found in the mobile application market with the concepts of service-orientation and the cloud, enabling customers, application vendors and providers to model, offer, customize and use such applications in a generic, easy and efficient way. Thus we advocate the industrialization of standard applications in a similar way as it is already done in other industries. Figure 1 shows the vision and main ideas we further describe in this paper. *Application vendors* develop applications that are then offered in an *application template catalogue* of the *application portal* of a *provider*. Therefore application vendors develop templates which are applications that contain open *variability* that must later be customized. Thus a *template package* format is needed as an exchange format between application vendors and providers. A template package must be able to contain code for the template, a *variability model* describing the variability as well as *requirements* on the provider infrastructure. To not limit application vendors in their choice of programming languages the package should be programming language agnostic and generic. We will show how these requirements regarding the modeling are met in Section III.

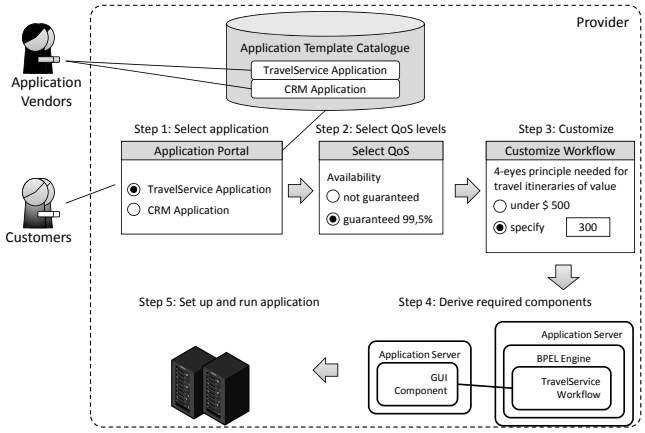

Figure 1. Overview

Once an application template for an application is offered at a provider, a *customer* can *subscribe* to such an application in five steps as shown in Figure 1. At first the customer *selects* an application template (step 1) then he *customizes*

functional and non-functional properties of the template (step 2 and 3). Afterwards the provider infrastructure takes over and *binds* necessary components and services needed to run the application (step 4) and then *provisions* the application (step 5) for the customer. The research questions arising from the customization and automatic provisioning of composite service-based applications are the following: How can customers be guided through complex customizations without having to know about implementation details? How can components be provisioned and configured automatically in the right order so that the whole application can later be run while respecting the selected functional and non-functional properties of the customer? These questions are addressed in terms of an architecture of the self-service portal for service-based applications developed in the Cafe[1] project, in Section IV. Concrete algorithms and tools that ensure the correct customization and provisioning are described in Section V. Thus, the main contributions of this paper are:

- A *generic metamodel* and package format for service-based cloud applications
- A *generic methodology* on how to define, customize and automatically deploy cloud applications from an application portal.
- A *reference architecture and implementation* for an application portal for service-based cloud applications.

By generic we mean that we do not impose any restrictions on basic frameworks, middleware or programming languages used in the applications to be offered in the portal. We evaluate the approach by presenting a variety of different applications which we have instrumented to be available in the portal in Section VI and finish with a summary and conclusion in Section VII.

II. Related Work

In this section we investigate related work and existing approaches. Application portals exist in various domains. In the mobile domain application stores exist for all major platforms (iPhone, Symbian, Android). However, applications must be specially developed for these portals and are then installed locally on the respective device. Other application portals such as the application portal of the Application Package Standard (APS) [3] offer the possibility to download Web-Applications that can then be installed using corresponding provisioning tools. As APS [3] is an application packaging standard for Web applications, it is limited to those applications and does not allow expressing complex variability dependencies and requirements on the provider infrastructure. In addition, it does not deal with the properties of (distributed) composite service-oriented applications such as the automatic configuration of service compositions during provisioning. Mashup tools [4] allow to compose Web applications out of a set of predefined components and run them, however, mashups are limited, cannot be moved between different platforms and are not suitable for enterprise applications in general. In [5] the self-serv environment for peer to peer applications is introduced. Our approach differs from that as it not only supports service compositions but whole applications including the automated provisioning of composed and aggregating services, and their variability. The service component architecture (SCA) [6] offers a model and package format for service-oriented applications. While individual components (services) of the application can be implemented in different programming languages (PHP, Java, C#, BPEL) SCA does not allow specifying requirements on the underlying infrastructure and requires an SCA compliant runtime. Also there is no possibility offered by the SCA specifications to describe complex variability points and their dependencies in an SCA application. The approach presented in [7] describes an approach to model complex application topologies including requirements for the underlying infrastructure. However, there is no possibility to define complex variability dependencies on all layers of the application and it is thus geared more towards system administrators / architects as it does not specify how customization can be done by end users. Similarly the approach in [8] does only allow describing requirements on the provider infrastructure as isolated non-functional requirements, whereas the approach presented here integrates the requirements into the variability model in order to describe complex requirement dependencies for functional and non-functional properties depending on customizations by the customer. In software product line engineering [9] applications can be annotated with variability models to describe customizable artifacts in an application. However, similar to [8] the focus lies on the customization and not the combination of customizability and automatic provisioning.

III. Modeling

In this section we describe our generic metamodels for application templates. An application template consists of an application model (which includes the files needed to run the application) and a variability model. The first metamodel is the application metamodel shown in Figure 2.

A. Application Metamodel

An *application model* contains a set of *components*. Components can be anything from hardware components over middleware components to components that realize application functionality such as a UI component, a workflow component a Web service or a database component. As application models are components themselves, they can again be recursively used in any other application model.

A component can have a *deployed on* relationship on one other component, meaning that it must be deployed on that other component. The set of components C and

[1] http://www.cloudy-apps.com

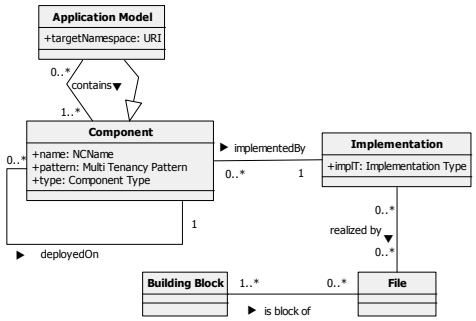

Figure 2. Application Metamodel

their deployed on relations $D \subseteq C \times C$ form a *deployment dependency graph* $DG = (C, D)$ which is directed and acyclic. Components have a unique name (unique in the application model) as well as a type (the *component type*). Additionally a *multi-tenancy pattern* is annotated to the component. Three multi-tenancy patterns are possible [10], [11], [8]: *Single instance*, i.e. the component can be shared between all customers (tenants) of the application. *Single configurable instance*, i.e. the component can be shared between all customers but every customer can customize certain aspects of the component and isolation of customers (tenants) is preserved. The last pattern is the *multiple instances* pattern, i.e. for each customer a new instance of the component is provisioned. Each component is implemented by an *implementation*. Zero or more *files* realize an application. Depending on the *implementation type* of the implementation, files can be of different nature. For example for the implementation type *Java Enterprise application* of a Java Web service, the file realizing the implementation could be an ear archive (.ear file). Thus, through the implementation type concept arbitrary programming languages or standards can be supported. However, it is not always desirable to include the files for the implementation of a component in the application model, for example for standard middleware. In this case the special implementation type *provider-supplied* can be used.

Provider-supplied indicates that the concrete implementation of a component must be provided by the provider which offers the application in his portal. For example, a Web service implemented as a *Java Enterprise application* realized by a .ear file can have a deployed on relationship (i.e. must be deployed) on an *application server* component of type *JBoss* that has an implementation with an implementation type of *provider-supplied*, which is not realized by any file, i.e. the provider must provision a *JBoss* application server on which the *Java Enterprise application* can then be deployed. Files consist of a set of *building blocks*. Building blocks do not need to be modeled explicitly as they are implicitly contained in files. Building blocks depend on the nature of the file, for example for an XML file they can be nodes in the XML tree, whereas for key,value based properties files they can be a key,value pair.

B. Variability Metamodel

Variability of service-based applications to be offered in an application portal is of essential importance to both customers and providers. Variability of a cloud application ranges from functional and non-functional variability in individual components (i.e. services, UIs or workflows) over variability in the composition (i.e. replacement of provided services with services already deployed in an enterprise) to variability (functional and non-functional) in middleware components and hardware components that must be supplied by the provider. As we aim at a generic solution and thus different components can have different implementation technologies an *orthogonal variability metamodel* [9] is needed allowing specifying variability across different components. As shown in Figure 3, a *variability model* built using our variability metamodel consists of a set of *variability points*. Each variability point represents a part of a component in an application model that can be customized. Exact locations inside the implementation files of a component are represented by building blocks. These can be referenced from a variability point by one or more *locators*. As we allow arbitrary implementation technologies, the locator concept needs to support various implementation artifacts and different types of building blocks. Therefore we, for example, introduce an *XPath locator* for XML documents which uses XPath expressions to locate a building block in a XML document. Another type, the *properties file locator*, points to a value of a (key,value) pair in a properties file. Other locators are possible for other implementation artifacts.

The customization options of a variability point are represented by *alternatives*. We say a variability point is *bound* when one of the alternatives is selected for the variability point during the customization. Different alternative types exist. *Free alternatives* allow a free value (restricted in its range or type by a *restriction*) to be entered to bind the variability point; *explicit alternatives* represent a predefined value that can be used to bind a variability point. An *expression alternative* contains an *expression* that computes a value that can be used to bind a variability point. This value can for example be computed based on the value entered at another variability point.

Variability points can depend on other variability points meaning that they must be bound after another variability point is bound. Variability points V and their *dependencies* $DP \subseteq V \times V$ form an directed acyclic *variability graph* $VG = (V, DP)$. In addition to the dependencies, *enabling conditions* can be added to a variability point that define which alternatives are enabled for selection (i.e., can be selected by a customizer). The enabling condition and dependency mechanisms allow defining complex dependencies between variability points in the form of "Alternative *A* can

be selected at variability point *VPA* if alternative *B* was selected at variability point *VPB*".

The *phase* of a variability point denotes in which phase of the provisioning of a new instance of the application it must be bound. Possible phases are:

- *Template customization* - i.e., before a provider offers the application in an application portal,
- *Solution engineering* - i.e., during customization by a customer,
- *Component binding* - i.e., when concrete components for provider-supplied components are searched,
- *Pre-provisioning* - i.e., before the provisioning of a component, and
- *Running* - i.e., after a component has been provisioned.

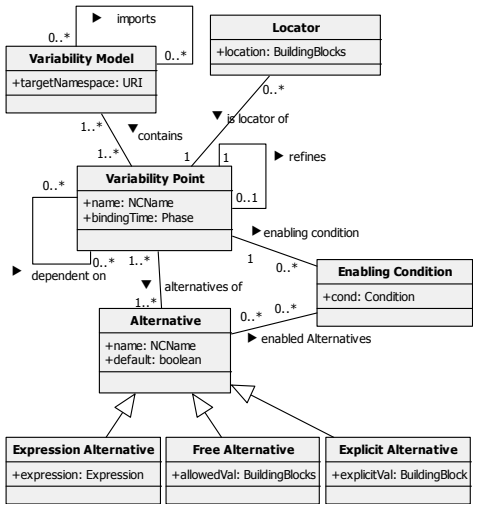

Figure 3. Variability Metamodel

Abstract Variability Models: Component types expose an abstract variability model which specifies the variability that must be bound if a component of that type is present in an application model. Thus a variability model for an application model implicitly *imports* the abstract variability models of the component types of components contained in the application model. For example, if an application component contains a component of type *JBoss*, the variability model for the application model implicitly imports all variability points associated with the component type *JBoss*, i.e. all these variability points must be bound before an instance of the application is usable by a customer. In case two components of the same component type exist in an application model their imported variability points are distinguished through different namespaces.

Variability Refinements: Imported variability points can be refined by other variability points. Refinement essentially means that the range of allowed values is restricted. For example an imported variability point *Version* that belongs to the abstract variability model of the component type *JBoss*, has three explicit alternatives with values 4.23, 5.01, and 5.10 and thus indicates that three different versions exist. However, in the application only version 5.10 can be used, thus the variability must always be bound with the alternative that represents version 5.10. Therefore the application vendor can *refine* the variability point *Version* in the variability model for the application by adding a new enabling condition that only allows this alternative to be selected. Other refinement possibilities include the addition of restrictions to free alternatives or the replacement of alternatives with other more strict alternatives.

Refined Variability Points as Requirements: The example of the refined *Version* variability point above already shows how refinements of variability points can be used to restrict the set of possible *realization components* of a provider-supplied component. Realization components are components that can either be newly provisioned or are already provisioned in the environment of the provider and can serve as concrete realizations of provider-supplied components. Besides the need of being of the same component type as the provider supplied component, such a component must also have the same multi-tenancy pattern assigned and its customization must be a correct and complete customization of the refinement of its abstract variability model as contained in the variability model of the application. Formally this means that all variability points for the component type that must be bound during the template customization phase must be bound with an alternative and value that is permitted in the refining variability model. For example, if a realization component of type *JBoss* has been "customized" to be (which in this case means "it is of") version 4.23 it is not a correct customization of a refined *Version* variability point that only allows selecting the alternative with value 5.10. Thus refined variability points of abstract variability models of component types constitute complex requirements on the underlying infrastructure. Note that using dependencies these requirements can be combined to form cross-component requirements (for example, "if component *A*'s *Location* variability point is bound to US, component *B*'s *Location* variability point cannot be bound to US"). This feature provides a more powerful configuration mechanism than the one presented in [7], [8] or the ones of APS [3] or SCA [6] where only local requirements can be expressed. In addition, because of the same metamodel for variability and requirements, choices made during the customization of the template such as selection of a certain quality of service by a customer, can influence the component selection afterwards, for example, whether a component in the local datacenter is used or one in a public cloud. Thus our generic orthogonal variability model is a powerful mechanism to describe both, variability and requirements on the provider infrastructure for an application template that goes beyond the variability models of other approaches while still being generic enough to cope with a variety of implementation technologies (see

also Section VI on the evaluation of the approach).

C. Package Format

Application templates are packaged in a specific package format so that they can be exchanged between application vendors and providers. This package format, named "Car" (composite application archive) is essentially a Zip file with a special META-INF directory that contains an XML serialization of the application model and the variability model. All implementation files referenced from the application model are also contained in that Zip file. We use a Zip file as most build tools such as Ant or Maven support the creation of such files.

IV. Runtime Architecture

Figure 5 shows the architecture of the application portal. The application vendor portal contains an *application template uploader component* where packaged application templates from application vendors can be uploaded. Application vendors use an *application modeler* tool that we implemented as a set of Eclipse plugins to model and package their applications according to the metamodels and package format defined above. The customer portal offers

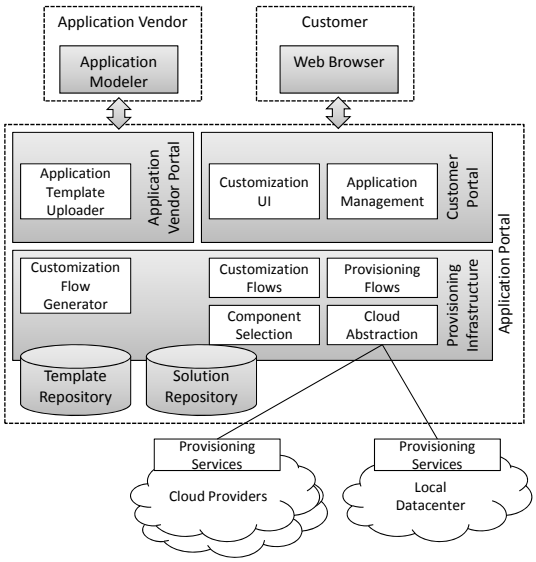

Figure 4. Runtime Architecture

customers a web-based self-service *application management* system. Customers can select, subscribe to, start, configure, manage and stop applications via their Web browser. When subscribing to an application a customer is directed to the *customization UI* which is a graphical front-end for the *customization flows* that guide a customer through the customization of an application template (see Section V). Once the customer has bound all variability related to the solution engineering phase, the variability is applied to the template and the resulting solution is stored in the *solution repository*. Then, similar to the very specific approach in [12] a generic BPEL *provisioning flow* executes and provisions the solution. How provisioning flows and customization flows are generated is described in detail in Section V. To understand the overall architecture it is sufficient to know that the provisioning flow makes use of the *component selection service*, which selects appropriate components for all provider-supplied components of the application. In addition it makes use of the *cloud abstraction*, which provides a unified interface for the provisioning services offered by the cloud providers, including interfaces for private clouds or provisioning engines in local data centers [13], [14]. These provisioning services are used to start and stop virtual machines, install software and gather configurations and monitoring data from the individual resources in the respective infrastructure. Once a provisioning flow has executed, the application is running and access details (in form of an URL) are given to the user in the application management UI. The whole application portal is built using a service-oriented architecture. In particular all components of the provisioning infrastructure are realized as services while the customization flows and provisioning flows are realized as service-orchestrations that themselves are recursively exposed as services for upper layers. This enables the distribution of the individual components (especially the provisioning services of different cloud providers) across different machines.

V. Implementation

In this section we describe the implementation of our prototype application portal in detail.

A. Customization Flows

Application vendors, providers as well as customers benefit from an easy, generic way to configure applications. We have introduced the orthogonal variability metamodel in Section III-B. As we have shown, these variability models can become fairly complex and can target implementation artifacts in a variety of different implementation techniques. However these technology details should be abstracted from the customers as they want to configure and set up applications in a fast and effective way without understanding the implementation details Thus we propose a *wizard-based* customization of applications that can be done by the customers in a self-service mode. However, these wizards must be developed by application vendors and thus impose additional work and costs. Therefore we follow the approach detailed in [15], [16] and make use of model-driven techniques to generate workflows that can be used to customize applications. We call these workflows *customization flows*. We have implementations of a BPEL generator that generates BPEL workflows including BPEL4People human tasks [16] and a case generator that generates cases for a case handling [17] system thus showing that the approach

works for a variety of workflow metamodels. The general algorithm behind the transformation of variability models into executable customization flows is the following (for details see [16]):

- Each set of variability points $VP(p)$ that must be bound in phase p is transformed into one workflow model WM_p.
- Each variability point $VP_i \in VP(p)$ becomes a *task scope* TS_i in the workflow model WM_p.
- A dependency between two variability points VP_i and VP_j is transformed into a control connector between the resulting task scopes TS_i and TS_j, thus dependencies are also respected in the resulting workflow model.
- For each enabling condition $ec_{i_1} \dots ec_{i_n}$ in a variability point VP_i, tasks $T_{i_1} \dots T_{i_n}$ are created in the workflow models that have an incoming transition condition that only evaluates to true if the respective enabling condition (and none of the previous enabling conditions) would enable to true. Thus, enabling conditions are respected.
- Each task T is either a *human task* if it requires human intervention (because a value must be entered or a choice must be made), or a *automatic task* that can be filled automatically by the customization flow.

One customization flow is generated for the template customization phase that can be used by the provider to customize a template to the providers capabilities. One customization flow is generated for the solution engineering phase to be used by the customer to customize an application and one for the component binding phase as well as one combined customization flow for the pre-provisioning and running phase. The latter two customization flows are used by the provisioning infrastructure to customize provisioning relevant variability points. How this is done is described in detail in the next subsection.

B. Provisioning Infrastructure

Once a customer has selected an application template from the application template catalogue as shown in Figure 5, the provisioning infrastructure takes over to provision the application. It performs four steps: It prompts the user for the customization, selects suitable components, computes the provisioning order and then provisions the components. These steps are detailed below:

Customization by the Customer: The first action the provisioning flow performs is to invoke the customization flow for the solution engineering phase. This customization flow then prompts the user for the necessary decisions in order to bind the variability. The provisioning flow then calls the solution creation service which is a Web service in the provisioning infrastructure that takes a template as well as its customization as an input and applies the customizations by interpreting the locators of the variability points and inserting the customized values at the right locations in the

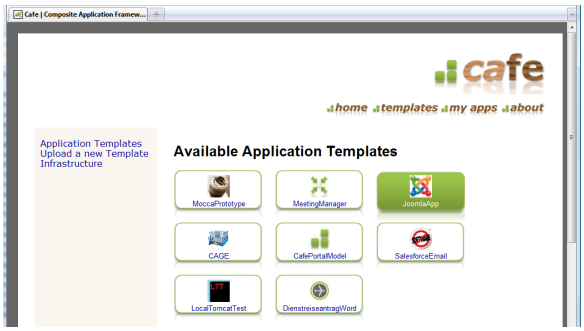

Figure 5. Application Template Catalogue

documents. It thereby first unzips zipped files such as .war files and zips them afterwards if files in such archives are targeted.

Selection of Suitable Components: Having created the customer-specific solution, the provisioning infrastructure must now select suitable realization components for those components that have an implementation type of provider-supplied. This cannot be done before as the customization might add, remove or modify components from the application model as well as modify requirements for provider-supplied components. Selection of suitable components is essentially an optimization problem with the goal to find the cheapest combination of already provisioned components and provisioning services that fulfills all implicit requirements and explicit requirements for all provider-supplied components in a solution. Implicit requirements are the component type and the multi-tenancy pattern, whereas explicit requirements are the requirements described using the variability refinements as shown in Section III-B. We do not go into details of the optimization problem here, however we have mapped this to a constraint optimization problem (COP) that can be solved using a standard COP solver. Having found the cheapest combination of realization components, the customization documents (i.e. the values of the bound variability points) of these realization components are used to bind the variability points that must be bound during customization binding.

Computation of the Provisioning Order: In parallel to the selection of the realization components, the provisioning flow computes the order in which the components of the solution must be provisioned. We define three ordering rules:

- **Rule 1:** Components can be provisioned after the component they must be provisioned on is running
- **Rule 2:** Components can be provisioned after all their pre-provisioning variability points are bound.
- **Rule 3:** Components must be provisioned in parallel where possible to speed up the whole process.

Rule 1 can be enforced by using the provisioning order graph as introduced in Section III-A and provisioning the leaf nodes first and then work up the graph. Rule 2 can

Pattern	Bound to	Provider-supplied	Action
Multiple Inst.	Provisioning Service	yes	provision
Single (Conf.) Inst.	Provisioning Service	yes	provision, subscribe
Single (Conf.) Inst.	Provisioned Component	yes/no	subscribe
All	-	no	deploy

Table I
PROVISIONING ACTIONS

be enforced by additionally taking the variability graph into account. Here it is necessary to investigate on which set of variability points VP_d the set of pre-provisioning variability points of a component C transitively depends. For each variability point in VP_d, it must be investigated whether this variability point is a running variability point and if it targets another component. If yes, component C must be provisioned after the other component as its variability points can only be bound after the other component has been provisioned. Rule 3 can be enforced by provisioning all components in parallel that do not have any dependencies as shown in Rule 1 and 2. Thus, to compute the provisioning order, the deployment graph (Rule 1) is enhanced with the relevant variability dependencies (Rule 2) into a *deployment dependency* graph. Note that this graph is always acyclic as the basic deployment graph is acyclic and we forbid adding variability point dependencies that would add cycles to the deployment dependency graph in the modeling tool. When reversing the edges of the deployment dependency graph the resulting *provisioning order graph* can be interpreted as a workflow that must be executed to provision the solution. Rule 3 is enforced as all components that do not have dependencies are automatically provisioned in parallel. In our implementation we execute a BPEL workflow that has the same semantics as the provisioning order graph and thus respects the dependencies and provisions components in parallel, where possible.

Provisioning a Component: When provisioning a component, the provisioning flow must respect the multi-tenancy pattern of the component and if it is realized by an already provisioned component or a provisioning service or if the implementation is included in the package (i.e. it is not of implementation type provider-supplied). Table I lists the different possibilities and the actions that need to be taken. These actions are described in detail below.

Provision: In this case a new component is provisioned by calling the provision operation of the associated provisioning service. We do not go into details here for space reasons, in brief that means sending a provision message to the Cloud abstraction layer including the customization document for all already bound variability points for the respective component. The cloud abstraction layer then translates the message into a sequence of necessary API calls for the underlying provisioning infrastructure. We have developed plugins for a Web hoster, Amazon EC2 as well as our local data center and thus can provision resources in any of these environments. The same holds true for the other actions that are mapped to other provisioning actions in the respective environment. The subscribe action is essentially a subscription to an existing multi-tenant aware resource or service. In case it is configurable, the necessary configuration data is submitted with the subscribe message. In particular for services in a service-based application that can be reused for different customers this action is important. The deploy action contains two steps, first of all the customizations made to the component that must be deployed are applied, i.e., the component is modified according to the pre-provisioning customizations. Then a deploy message is sent to the cloud abstraction layer that contains an EPR under which the component code can be downloaded. Then the component abstraction layer calls the necessary provisioning actions (i.e. SCP-based file upload and execution of commands via SSH for Amazon EC2) to deploy the component. While provisioning, the provisioning infrastructure also automatically binds the variability points that must be bound during the pre-provisioning and running phase of a component. Such variability points can be for example EPRs or locations of newly provisioned services.

VI. EVALUATION

In this section we report on the evaluation of the general applicability of our approach. This is done by showing how the approach can be used in various scenarios. As we have said in the introduction in Section 1 the main contribution of this paper is a generic framework to model, customize and execute applications in the cloud. In Table 2 we show some of the sample application templates that we have modeled and packaged using the metamodels and package format described in Section 4. Table 2 also shows the implementation technologies and the runtime environments on which the respective applications can be provisioned. We uploaded these application templates to our application portal. Users can now select applications and provision them automatically. Components are provisioned on EC2 in the local data center, at a Web hoster and on the local machine that runs the portal. As shown in Table II different programming languages and technologies are used such as PHP, Java, Ruby, SQL, BPEL, Apache, Apache Tomcat, MySQL, HSQLDB, JBoss, Tomcat, Apache ODE, Genesis [18], ActiveBPEL and one application even consists out of a Word file that is customized by a customization flow and then provisioned on an Apache Webserver where it can then be downloaded. The number of components ranges from 2 in the Travel Approval example to over 20 in the Order Processing example, provisioning times (after customization) heavily depend on the number of components and the target environment and range from around 25 seconds in the Travel

Application	Implementation Technologies	Runtime Environment
Joomla CMS	HTML, PHP, MySQL, Apache	Web Hoster, EC2
Order Processing	JEE 5, HSQLDB, JBoss, Tomcat	EC2, local machine
Salesforce SMS Sender	Java 6, BPEL, Tomcat, Apache ODE	EC2, Salesforce, local datacenter
CAGE SOA Test	Ruby, BPEL, Apache ODE, Genesis	EC2
Application Portal	JEE 5, MySQL, BPEL, JBoss, ActiveBPEL	EC2, local machine, local datacenter
Travel Approval	Word, Apache	Local machine

Table II
EXAMPLE APPLICATIONS

Approval example to several minutes in the Application Portal and Order Processing Examples. One interesting note is that we can also bootstrap our Application Portal as we have modeled and packaged it as shown above and can now provision it from another application portal.

VII. SUMMARY AND OUTLOOK

In this paper we presented a generic approach for the modelling, packaging, customization and on-demand provisioning of composite service-based applications. We have shown a metamodel on how to model such applications, we have introduced a variability metamodel to model customizable aspects of the application, and have shown how variability models can be transformed into workflows that can be used by customers and the provisioning infrastructure to customize the applications. Despite allowing a wide range of programming languages our approach allows customers to start, configure and stop applications in a self-service portal, without having any knowledge about the implementation of applications. We have shown how such applications can be provisioned automatically and have evaluated the general applicability by reporting on several case studies where we have applied the approach and technologies. In future work, we will extend the approach to include aspects of dynamic provisioning and monitoring to dynamically scale applications after they have been provisioned. We are also in the process of deriving more efficient optimization algorithms for the component binding as well as investigate function shipping for components or minimizing of communication overhead when distributing components across clouds.

REFERENCES

[1] M. Armbrust *et al.*, "Above the clouds: A berkeley view of cloud computing," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-28*, 2009.

[2] W. Sun, X. Zhang, C. Guo, P. Sun, and H. Su, "Software as a service: Configuration and customization perspectives," in *IEEE SERVICES-2*, 2008.

[3] SWSoft Inc., "Application Packaging Standard (APS)," http://apsstandard.com/r/doc/package-format-specification-1.0.pdf, 2007.

[4] J. Yu, B. Benatallah, F. Casati, and F. Daniel, "Understanding mashup development," *IEEE Internet Computing*, vol. 12, no. 5, pp. 44–52, 2008.

[5] B. Benatallah, Q. Sheng, and M. Dumas, "The self-serv environment for web services composition," *IEEE Internet Computing*, vol. 7, no. 1, pp. 40–48, 2003.

[6] Open SOA Collaboration (OSOA), "SCA Service Component Architecture, Assembly Model Specification Version 1.00," http://www.osoa.org/download/attachments/35/SCA_AssemblyModel_V100.pdf, 2007.

[7] W. Arnold, T. Eilam, M. Kalantar, A. V. Konstantinou, and A. A. Totok, "Pattern based soa deployment," in *Proc. of ICSOC '07*. Springer-Verlag, 2007.

[8] T. Unger, R. Mietzner, and F. Leymann, "Customer-defined Service Level Agreements for Composite Applications," *Enterprise Information Systems*, vol. 3, no. 3, 2009.

[9] K. Pohl and A. Metzger, "Variability management in software product line engineering," in *Proceedings of the 28th international conference on Software engineering*. ACM, 2006.

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

[11] C. Guo, W. Sun, Y. Huang, Z. Wang, B. Gao, and B. IBM, "A framework for native multi-tenancy application development and management," in *CEC/EEE 2007*.

[12] A. Keller and R. Badonnel, "Automating the Provisioning of Application Services with the BPEL4WS Workflow Language," *Proc. DSOM 2004*, 2004.

[13] 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 *Services-Part I, 2008. IEEE Congress on*. IEEE, 2008, pp. 3–10.

[14] R. Mietzner, T. Unger, and F. Leymann, "Cafe: A generic configurable customizable composite cloud application framework," *On the Move to Meaningful Internet Systems: OTM 2009*, pp. 357–364, 2009.

[15] M. Mendonca, D. Cowan, and T. Oliveira, "A Process-Centric Approach for Coordinating Product Configuration Decisions," *In Proc. HICSS 2007*, 2007.

[16] R. Mietzner and F. Leymann, "Generation of BPEL Customization Processes for SaaS Applications from Variability Descriptors," in *Proceedings of the International Conference on Services Computing, SCC 2008*. IEEE, 2008.

[17] W. Van der Aalst, M. Weske, and D. Grünbauer, "Case handling: a new paradigm for business process support," *Data & Knowledge Engineering*, vol. 53, no. 2, pp. 129–162, 2005.

[18] L. Juszczyk, H. Truong, and S. Dustdar, "Genesis-a framework for automatic generation and steering of testbeds of complexweb services," in *13th IEEE International Conference on Engineering of Complex Computer Systems*, 2008.