Cafe: A Generic Configurable Customizable Composite Cloud Application Framework*

Ralph Mietzner, Tobias Unger, and Frank Leymann

Institute of Architecture of Application Systems University of Stuttgart
Universitaetsstr. 38 70569 Stuttgart, Germany
firstname.lastname@iaas.uni-stuttgart.de
http://www.iaas.uni-stuttgart.de

Abstract. In this paper we present Cafe (Composite Application Framework) an approach to describe configurable composite service-oriented applications and to automatically provision them across different providers. Cafe enables independent software vendors to describe their composite service-oriented applications and the components that are used to assemble them. Components can be internal to the application or external and can be deployed in any of the delivery models present in the cloud. The components are annotated with requirements for the infrastructure they later need to be run on. Providers on the other hand advertise their infrastructure services by describing them as infrastructure capabilities. The separation of software vendors and providers enables end users and providers to follow a best-of-breed strategy by combining arbitrary applications with arbitrary providers. We show how such applications can be automatically provisioned and present an architecture and a prototype that implements the concepts.

1 Introduction

Driven by the need to outsource (part of) their IT-infrastructure, companies have shifted considerable amounts of their IT from their own premises to external companies. Therefore new delivery models for software have been emerging that allow the outsourcing of different aspects of an application. The software as a service (SaaS) model is a prominent example of such an outsourcing model. Other delivery models such as Infrastructure as a Service (IaaS) and Platform as a Service (PaaS) aim at providing (parts of) the necessary infrastructure and platform support to easily built and host applications at a provider.

The advent of these delivery models and new architectural styles such as service-oriented architecture (SOA) and emerging technologies such as Web services, virtualization and the abundance of fast internet connections has lead to new ways to deliver software. Service-oriented applications can now be assembled out of services that are running in one's own infrastructure as well as at

* This work is partially funded by the EU 7th Framework Project ALLOW (http://www.allow-project.eu/) (contract no. FP7-213339).

R. Meersman, T. Dillon, P. Herrero (Eds.): OTM 2009, Part I, LNCS 5870, pp. 357–364, 2009.

third parties and can be offered "as a service" to multiple customers. However, current approaches for modeling service-based applications often neglect the aspects concerning the modeling of requirements of an application on the runtime infrastructure or modeling of the runtime infrastructure itself. This is due to the fact that traditionally, applications are often developed with certain knowledge about the runtime infrastructure on which they are later deployed. In the new delivery models, the runtime infrastructure is (partially) outsourced to "the cloud" and the application must be automatically deployed at one or several providers. Therefore applications must be developed independently of the provider infrastructure they are later run on. This enables their automated provisioning at and across different providers. In this paper we introduce Cafe (composite application framework) a model and provisioning infrastructure to describe and provision composite, service-oriented applications and their required infrastructure. We start the paper with a motivating scenario in Section 2. In Section 3 we then briefly recapitulate how customizable composite applications and their requirements for the infrastructure can be modeled using so-called component and deployment graphs as well as variability descriptors. We show how different delivery models can be integrated for different components. We then show in Section 4 how the model can be used to automatically provision applications across different providers. We finish the paper by comparing our approach to related work and by giving hints for future work.

2 Periodic Inspection Reminder Scenario

In Germany, passenger cars must be safety-inspected every two years. As customers often forget that an inspection is required, the car dealer "Perfect Cars" wants to offer a new service to its customers, which reminds them via a SMS message or E-Mail when an inspection is required.

The inspection reminder software from "Perfect Software" (cf. Figure 1) provides a Web interface to the car dealer for triggering inspection reminders. Customer data is stored in a customer relationship management system (CRM service) that is accessed by the application. The CRM service can either be the one internal to the application or the Salesforce[1] Web service. In case the Salesforce Web service is used, a Salesforce wrapper component is to access the Salesforce API. Another component of the application is an E-Mail service that sends inspection reminders, this internal E-Mail service can be substituted for an external SMS service. The CRM service and SMS/E-Mail services are orchestrated by a BPEL workflow that implements the core application logic (reminder workflow in Figure 1). "Perfect hosting" is an application hosting company that offers the inspection reminder application they bought from "Perfect Software" to customers "as a service". New tenants (such as "Perfect Cars") can subscribe to the application via the application portal of perfect hosting.

[1] http://www.salesforce.com

3 Application Model

In this section we briefly introduce the formal concepts that are necessary to understand the algorithms in Section 4. A full formalization can be found in [9].

Applications $\mathcal{A}$ are assembled out of a set of *components* $\mathcal{C}$. Components can be application components (such as UIs, services and workflows) and infrastructure components (such as servers or middleware). These components are wired together via a set of *wires* $\mathcal{W}$. A wire between two components denotes a directed usage link. In addition to wires, the model allows to describe a set of *deployment relationships* $\mathcal{D}$. A deployment relationship between two components c_1 and c_2 denotes that the first component (c_1) is deployed on the second component (c_2).

Definition 1 (Applications). *The set of applications is defined as:*

$$\begin{aligned}\mathcal{A} &= \{a_1, \ldots, a_n\} \\ &= \{(C, W, D) | C \subseteq \mathcal{C} \wedge W \subseteq C \times C \subseteq \mathcal{W} \\ &\quad \wedge D \subseteq C \times C \subseteq \mathcal{D}\}\end{aligned}$$

We call the components and their wires the *component graph*. This is a directed, possibly cyclic graph. Besides communicating with each other (expressed in our model through wires) components may have a second relationship: the deployment relationship. We call the components and their deployment relations the *deployment graph*. The deployment graph is a directed, acyclic graph.

Components may be associated with certain properties. In the following we introduce the properties of a component that we need for the algorithms below, for additional properties see [9]: The function $implType : \mathcal{C} \rightarrow \mathcal{I}$ assigns an *implementation type* to a component. A member of the set of implementation types $\mathcal{I}$ can be any programming language. For example a workflow component could be implemented in BPEL. It can also be "providerSupplied" to denote that the provider must somehow provide the component.

Each component has an assigned *multi-tenancy pattern*. The multi-tenancy pattern can either be: (i) *single instance* - i.e. the component is shared by multiple

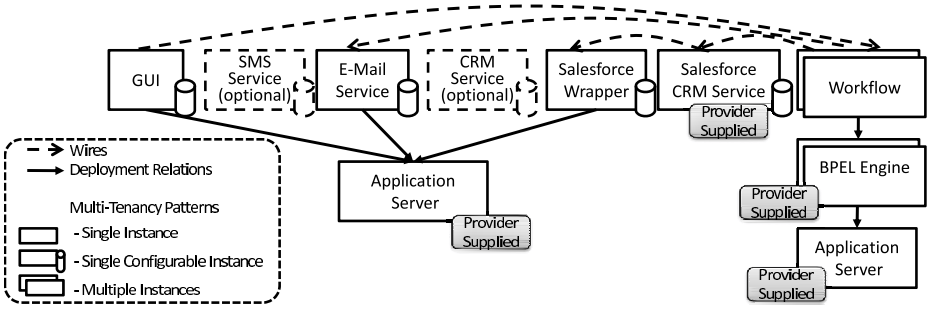

Fig. 1. Application Model

tenants, (ii) *single configurable instance* - i.e. the component is shared by multiple tenants and must be configured for every tenant, or (iii)*multiple instances* - i.e. the component must be deployed separately for each new tenant. The function $pattern : C \rightarrow \{singleInstance, singleConfInstance, multiInstances\}$ assigns a multi-tenancy pattern to a component.

Components can have requirements on other components (i.e. an application component can have a requirement on a middleware component it needs to be deployed on. A requirement has a type (e.g. availability or response time). For a discussion and formal representation of requirements and their types see [9].

4 Application Provisioning

During the deployment of a new tenant for an application four main steps are taken. In the first step the user selects a template application from a directory. In the second step the tenant customizes the application through a so-called *customization workflow* (for a discussion of these steps see [4]). In the third step the components required to run the new tenant are mapped to the already available environment and in the fourth step those application components that are not already available are provisioned. During customization, components might be added or removed from the template component and deployment graphs of the application. Additionally, alternatives are bound. For example the tenant selects that he wants to use the E-Mail service and the Salesforce CRM application in our sample application shown in Figure 1. Therefore the SMS service and the internal CRM service are removed from the component graph and deployment graph. Thus, customization can be a modification of the component and deployment graphs.

4.1 Matching Step

Once an application is ready to be provisioned for a tenant the first step is to examine which components of the application are already deployed for other tenants and can be reused. Therefore the deployment graph of the application is traversed top down. I.e. for each *root component* of the application it is checked whether a suitable match can be found in the environment. A root component is a component that has no incoming deployment relationship. We can restrict this step to the matching of components in either the single instance or single configurable instance patterns. Multiple instance components must be provisioned anyways so no matching is needed. Algorithm 1 shows the whole algorithm.

A suitable match is found if at least one component of the same type as the component in the application exists in the environment that has capabilities that match the requirement annotations of the component in the application.

The set of matching components is determined using the *match* function. The set $matchingComponents(c) = \{d \in \mathcal{CA} : match((c, d)) = true\}$ is defined as the set of components available at any provider $\mathcal{CA}$ that match the component

c. The function $match : \mathcal{C} \times \mathcal{CA} \rightarrow \{true, false\}$ is used in order to determine whether a component c in an application can be matched to a component or sub-graph represented through it's root component d at a provider. I.e., the application can be configured to use that component d.

In [6] we describe how policies expressed in WS-Policy can be matched to properties of resources in an environment. These algorithms can be used as the capabilities as well as the requirements for components can be described as policies. Other matching techniques such as semantic matching of services can be also employed to find suitable available components that can be assigned to components in an application.

Having found several possible matches, one of these matches must be selected. This can for example be done based on performance or cost. We will present different optimization algorithms that we have investigated in future work. For now we introduce a *selectBest* function that encapsulates the selection. In our prototype we simply take the first component out of the set of matching components. Formally the *selectBest* function is defined as: $selectBest : 2^{\mathcal{C}} \rightarrow \mathcal{C}$ The function $matchedComponent : \mathcal{C} \rightarrow \mathcal{CA}$ assigns exactly one component out of the set of available components to a component.

In case no matching is found for a component all the "child" components of this component are examined and matched. We define the set of "child" components $children(c)$ of a component $c \in C_a$ as the set of those components of application $a = C, W, D$ on which c has a deployment relationship. $children(c) = \{c_1 \in C_a | \exists d \in D : \pi_1(d) = c \wedge \pi_2(d) = c_1\}$.

In case a component is a child of multiple other components, such as the application server in our example, this component is matched the first time one of it's parent components cannot be matched. The next time the previous matching is reused.

Once the children of a component have been matched (or a suitable provisioning service has been found) a suitable *provisioning service* for the component needs to be found. A provisioning service [5] is a service that can provision a certain set of components. The available provisioning services at all providers is described through the set PS. We do not go into details here, on how to find such services or how a registry for different providers of such services can be built. In [6] we describe how WS-Policies can be used to find suitable Web services that describe their non-functional properties with WS-Policies. This work can be reused here. The function $findProvisioningService : \mathcal{C} \rightarrow PS$ searches for a suitable provisioning service for a certain component c A suitable provisioning service is a service that can provision the component or the virtual leaf graph with the necessary quality of service. The function $provisioningService : \mathcal{C} \rightarrow PS$ assigns a provisioning service to a component. Implicitly we assume that a component that can be matched will never be redeployed. However, there might be situations where it is cheaper to redeploy such a component (for example because a new provider offers a cheaper component than the one used before). Such a component can be treated as a component that could not be matched.

Algorithm 1. Matching Algorithm

```
1: {Algorithm is started with the set of root components RC}
2: function findMatch (Components)
3: {examine all components in the set}
4: for all c ∈ Components do
5:    { check if the component is in the single instance or single configurable instance
      pattern and there exists a match in any of the available infrastructures }
6:    if (pattern(c) ∈ {singleInst, singleConfInst}) ∧ ((matchingComponents(c)! =
      ∅) then
7:       {select the best matching component for c}
8:       matchedComponent(c) = selectBest(matchingComponents(c))
9:    else
10:      {no matching component found, examine the underlying components }
11:      findMatch(children(c))
12:      {after all children have been treated - find a suitable provisioning service}
13:      provisioningService(c) = findProvisioningService(c)
14:   end if
15: end for
16: end function
```

4.2 Configuring the Properties of Components

After the matching of components to actual providers, components that use other components (i.e., components that are the source of a wire) must be configured with the properties (e.g. the EPR) of the component the wire points to. Two situations can arise now: (i) The property value is already available because the component is already deployed or the property is static. (ii) The property becomes available after provisioning. In case (i) the source component can be configured and then provisioned as described in the next section. In case (ii) the source component must be provisioned after the target component has been provisioned. Therefore the deployment step must take this ordering into account. This is done by reordering sibling components in the deployment graph. Reordering is not always possible, for example if two components have a cyclic wire dependency. In this case two solutions can be employed: (i) One of the components can be configured after deployment, then, this one is provisioned first. (ii) A mediator component (such as a virtual endpoint at an ESB for endpoint resolution) can be put between those components and can be configured with the concrete properties later. The components are then configured to use the mediator component.

4.3 Deployment of Components

Having obtained the properties of all components, the components that are not already matched can be provisioned, and those that are already matched can be configured. In our prototype we use a BPEL provisioning flow that traverses the deployment graph depth-first (taking the ordering of siblings into account that

has been determined based on the wires) and calls the necessary provisioning services (that can be at different providers).

Using a full-blown workflow system such as a BPEL engine has the advantage over other scripting languages that the features of BPEL such as transactionality, human tasks (via BPEL4People), Web service invocations directly out of the process can be reused. In [5] we describe the architecture for a unified provisioning infrastructure that spans across multiple providers and that can be used to deploy the applications described in this paper.

In short the algorithm calls the provisioning service for each component that is not already matched and provisions this component. Components that are already matched and are deployed in the single configurable instance pattern need to be configured for the new tenant. In this case the "addTenant" method of the component is called that allows to deploy new configuration data for a new tenant. Therefore all components that can be deployed in single configurable instance mode must provide such an "addTenant" method. In [5] we describe how such methods can be provided on top of existing provisioning engines.

5 Related Work

The authors of [1] describe a framework for the configuration, distribution and deployment of Web services. They describe how Web services based applications can be configured and deployed. Our work differs from their approach as it allows to explicitly model multi-tenancy in the application. Additionally our model allows to model requirements on the necessary infrastructure for components. In [8] the provisioning and adaptation of composite Web service based applications is discussed with regards to hybrid environments. This approach nicely complements our approach as it allows providers to optimize the allocation of services to resources. Therefore this approach can be seen as one strategy for providers to provision an application in our scenario. However, the approach must be extended by the notion of multi-tenancy to be fully usable in our scenarios which also holds true for the SLA-driven provisioning approach presented in [3]. In [7] the Vienna component framework is presented that allows to combine components from different component models into a new application, however it does not include multi-tenancy patterns and different delivery models as well as the modeling of the required infrastructure. In [2] a framework for the deployment modeling of (SOA) applications is presented that is similar to our approach. While the approach presented in [2] allows to explicitly model the deployment relationships between infrastructure components down to the bare metal level (although you do not have to do this) we delegate this to the provider. Thus our approach is more focused on cross-provider based applications. Also the variability mechanisms in [2] differ from ours as they rely on templates that describe components or deployments that can then be customized. We focus on the variability in the application and the component graph and our orthogonal model allows to combine both. The topology definition described in [2] could be used as an input for a highly configurable provisioning service.

6 Conclusions and Future Work

In this paper we presented Cafe a model and provisioning architecture on how to specify composite applications that can be offered on demand by different providers. We introduced a component graph as a graph-based structure to capture the dependencies between components . A so-called deployment graph has been introduced to capture deployment relationships between components. We used these structures to show how components can be matched against the infrastructures offered by different providers. We showed how components that are not available can be automatically provisioned and how various delivery models can be integrated in the approach. In future work we will show how the distribution of components at one provider or across different providers can be optimized regarding costs for the running of the application. We presented the necessary formalisms based on the deployment and component graphs to describe how the matching and provisioning algorithms work in detail. We also showed how variability is captured in the model and how it affects the deployment of components. We have implemented a prototype that serves as a proof of concept of the conceptual work presented before.

References

1. Anzböck, R., Dustdar, S., Gall, H.: Software Configuration, Distribution, and Deployment of Web-Services. In: SEKE (2002)
2. Arnold, W., Eilam, T., Kalantar, M.H., Konstantinou, A.V., Totok, A.: Pattern based soa deployment. In: Krämer, B.J., Lin, K.-J., Narasimhan, P. (eds.) ICSOC 2007. LNCS, vol. 4749, pp. 1–12. Springer, Heidelberg (2007)
3. Ludwig, H., Gimpel, H., Dan, A., Kearney, B.: Template-Based Automated Service Provisioning-Supporting the Agreement-Driven Service Life-Cycle. In: Benatallah, B., Casati, F., Traverso, P. (eds.) ICSOC 2005. LNCS, vol. 3826, pp. 283–295. Springer, Heidelberg (2005)
4. Mietzner, R., Leymann, F.: Generation of BPEL Customization Processes for SaaS Applications from Variability Descriptors. In: SCC 2008 (2008)
5. Mietzner, R., Leymann, F.: Towards Provisioning the Cloud: On the Usage of Multi-Granularity Flows and Services to Realize a Unified Provisioning Infrastructure for SaaS Applications. In: SERVICES 2008 (2008)
6. Mietzner, R., van Lessen, T., Wiese, A., Wieland, M., Karastoyanova, D., Leymann, F.: Virtualizing Services and Resources with ProBus: The WS-Policy-Aware Service and Resource Bus. In: Proceedings of the 7th International Conference on Web Services, ICWS 2009 (2009)
7. Oberleitner, J., Gschwind, T., Jazayeri, M.: The Vienna Component Framework enabling composition across component models. In: Proceedings of 25th International Conference on Software Engineering, pp. 25–35 (2003)
8. Sheng, Q.Z., Benatallah, B., Maamar, Z., Dumas, M., Ngu, A.H.H.: Enabling Personalized Composition and Adaptive Provisioning of Web Services. In: Persson, A., Stirna, J. (eds.) CAiSE 2004. LNCS, vol. 3084, pp. 322–337. Springer, Heidelberg (2004)
9. Unger, T., Mietzner, R., Leymann, F.: Customer-defined Service Level Agreements for Composite Applications. Enterprise Information Systems 3(3) (2009)