Defining Composite Configurable SaaS Application Packages Using SCA, Variability Descriptors and Multi-Tenancy Patterns *

Ralph Mietzner, Frank Leymann
University of Stuttgart
Institute of Architecture of Application Systems
Universitaetsstr. 38
70569 Stuttgart, Germany
{Mietzner, Leymann}@iaas.uni-stuttgart.de

Mike P. Papazoglou
Tilburg University
Dept. of Information Systems & Management
P.O. Box 90153
5000 LE Tilburg, The Netherlands
mikep@uvt.nl

Abstract

Currently, several vendors and projects are building proprietary SaaS platforms where more and more applications are hosted in a Software as a Service business model. However, these proprietary platforms prevent that applications offered by different SaaS application vendors can be easily reused on the platforms offered by the different SaaS hosting providers. In this paper we describe a package format for composite configurable SaaS application packages for applications developed following a service oriented architecture. We show how the service component architecture (SCA) can be extended with variability descriptors and SaaS multi-tenancy patterns to package and deploy multi-tenant aware configurable composite SaaS applications.

1. Introduction

More and more software vendors introduce software as a service (SaaS) delivery models for their software. Software as a Service is a delivery model for software where software is no longer purchased and run by customers themselves on their own infrastructure but is run on the IT infrastructure of a hosting company. To address this problem, we introduce three roles in a typical SaaS scenario. The first role is the SaaS customer. The SaaS customer is the person or company that wants to use software to fulfill a certain (business) task and therefore subscribes to a SaaS application. The second role is the SaaS provider. The SaaS provider sells software as a service. The SaaS provider therefore runs the software to which SaaS customers can subscribe. The third role in the SaaS model is that of the SaaS application vendor. SaaS application vendors are companies that develop the applications that are offered as a service by SaaS providers. SaaS providers and SaaS application vendors can be the same companies. For example Salesforce.com [7] offers their CRM application as a service therefore acting as a SaaS provider as well as a SaaS application vendor. Additionally Salesforce.com allows other companies to develop SaaS applications and run them on the Salesforce platform thus acting as a SaaS provider that hosts the applications of the third party SaaS application vendors. The SaaS provider and the SaaS customer, must not even be different companies. For example the IT division of a big company may offer software as a service to other divisions of that company in an "intra-enterprise SaaS" business model. In a recursive outsourcing model SaaS providers themselves can outsource (parts of) the hosting of a SaaS application to third parties that host the applications for them. A few important requirements must be met to bring together SaaS providers as well as SaaS application vendors in a way that a SaaS provider can simply take an application developed by a SaaS application vendor and run it in its data center: The application must be packaged in a format that can be easily understood and exchanged between SaaS application vendor and SaaS provider. The package format must contain all the necessary artifacts that are needed to provision the application in the SaaS provider's data center. The application must identify so-called *variability points* [1] where customers can customize the application according to their needs. Therefore two types of application packages are needed. The first package is the one that is provided by the SaaS application vendor. It contains variability points that specify where customers can customize the application and therefore constitutes a template for the application. The second package is the customized application derived from the application template. It is specific to one customer. We call this second package a *solution package*. The solution pack-

*The research leading to these results has received funding from the European Community's Seventh Framework Programme under the Network of Excellence S-Cube - Grant Agreement no. 215483.

978-0-7695-3163-2/08 $25.00 © 2008 IEEE
DOI 10.1109/ICIW.2008.68

age can be deployed in the data center of an SaaS provider. Another requirement for SaaS applications is the support for multiple tenants. A tenant in a SaaS scenario is a SaaS customer that uses a SaaS application. In order to exploit economies of scale, i.e. allow a SaaS provider to offer the same instance of a SaaS application to multiple tenants, the application must be multi-tenant aware. Multi-tenant aware means that each tenant can interact with the application as if he were the sole user of the application. In particular a tenant cannot access or view the data of another tenant. In order to allow the configuration of multi-tenant aware applications on a per-tenant basis, the application must be divided in two different parts. The first part of the application describes the artifacts that are fixed and can be used by all tenants whereas the second part describes what [2] call "configuration metadata". Data that is tenant specific and must be deployed with every new tenant. In this paper we introduce a package format for application templates and solutions based on the service component architecture (SCA) [5] standard. We describe a way to annotate SCA with variability descriptors in order to explain how and where the application can be customized. Additionally we introduce SaaS multi-tenancy patterns as a way to annotate SCA artifacts with deployment information needed to generate automatic solution packages for both, the basic application as well as the tenant-specific parts of the application.

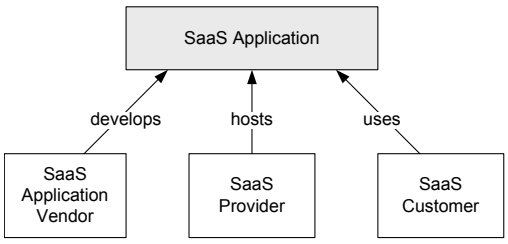

Figure 1. Roles in a SaaS scenario.

2. Service Component Architecture

The Service component architecture (SCA) is a set of specifications describing a package format for applications built following a service oriented architecture. Other specifications describing package formats for enterprise applications include Java EE .ear files which focus on Java applications or the Application Packaging Standard [8] which focuses on the delivery of Web applications. We focus on the SCA standard in this paper since it allows to specify composite SOA applications developed in different programming languages and already includes a basic mechanism for configurability of the components that form such an application. We describe the main building blocks of SCA in order to motivate its benefits and limitations for SaaS application packages. The SCA Assembly Model Specification [5] specifies how an SCA composite application (called *composite*) is built from individual artifacts (called *components*). An SCA component represents a business function. A component provides *services* that are used to call this component. Components can call other components thus consuming their services via *references*. A component reference is connected to a component service via a *wire* which points to a service. As components, composites can contain services and references. Services of components contained in the composite can be promoted as a service of the composite that can be called by requestors from outside the composite. Similar to component services, component references can be promoted to composite references. Composite references denote that the reference is served by a service outside the composite. A composite including all its components is described in an XML document that is specified by the SCA assembly model specification. We will call this document the *composite descriptor* from now on. An SCA component is a configured instance of an SCA component implementation. This means that several components can use the same implementation but configure it differently. A component implementation is a concrete implementation of a business function. SCA provides a set of predefined implementation types that refer to specific implementation technologies such as Java, BPEL, C++, Spring or Java EE EJB technology. The SCA assembly model is recursive by allowing composites to be implementations of components in other composites. Implementations (including composites) can have properties. Using properties an implementation can be configured externally by specifying property values in the component that uses an implementation. In particular different components can use the same implementation but configure it differently by specifying different property values. For example component A specifies that the implementation should use table A in the database while component B specifies that the implementation should use table B. Properties can be accessed in the implementation code. SCA specifies how properties are specified and accessed for example in Java and BPEL code. The SCA assembly model specification also describes how the SCA and non-SCA artifacts (such as code files) are packaged and deployed. The central unit of deployment in SCA is a contribution. A contribution is a package that contains implementations, interfaces and other artifacts necessary to run components. The SCA specification specifies an interoperable package format for contributions. The package format proposed by the SCA assembly model specification is a Zip file, however other packaging formats are explicitly allowed. The SCA specification does not impose any restrictions on these other than it must be a directory-like structure with a single root in which a META-INF directory exists in which a `sca-contribution.xml` file must be placed.

The `sca-contribution.xml` file lists the SCA composites within the contribution that are runnable, i.e., that can be installed in the SCA domain the contribution is installed into. An SCA domain represents a complete runtime configuration, potentially distributed over a series of runtime nodes. Additionally an SCA domain defines the boundary of visibility for all SCA mechanisms [5]. In particular this means that artifacts (such as services) from one contribution are visible to artifacts (such as a reference) from another contribution if both contributions reside in the same SCA domain and are made explicitly visible through so-called imports and exports.

3. Using SCA as a Configurable SaaS Application Package Format

In this section we describe how SCA can be used as an application template package format for SaaS application templates and solutions. In order to evaluate which changes or additions are needed to SCA we need to recapture the requirements for application packages we described in the introduction. The first requirement was that the package must describe and contain all artifacts needed for the application. We can use SCA's packaging mechanism and consider one SaaS application as a SCA contribution that contains all the necessary artifacts (such as code files, interface files, etc.). The second requirement is that variability points must be described in the package. SCA offers a mechanism to specify variability of implementations through properties. Each component can define properties for its implementations that can be set outside of the implementation in the component declaration in the composite descriptor. However, the properties are only a rudimentary configuration mechanism. They lack the possibility to specify alternative values, constrained values and dependencies between properties that are needed to specify complex configuration possibilities. [3] For example it is not possible to specify dependencies between properties such as if property A is filled with the value X the property B might not be filled with the value Y.

3.1. Adding Variability to SCA packages

In order to fulfill the requirements for the description of variability in application templates we propose to extend the SCA properties mechanism with the use of variability descriptors [4]. These are used to describe complex configuration options. A variability descriptor consists of a set of variability points. A variability point describes the location of variability in a component, i.e. it contains a locator that points into a location inside a document (such as an activity in a BPEL [6] file or a property in a SCA composite descriptor). The variability point then defines the values that are permitted for that variability point through alternatives. Alternatives can describe fixed values (in form of explicit alternatives) or expressions. These are described in the form of expression alternatives that can also point into the same or into another document, empty alternatives, which specify that the variability point can be simply omitted and free alternatives that specify that a user must enter a value. Free alternatives can be constrained by expressions that specify which values are allowed. Text and especially XML files contained in a contribution can be directly annotated with variability points through variability descriptors. However, SCA's property mechanism can be reused and the variability points can point to the declaration of the values for the property mechanism in the composite descriptor. This has the advantage that the actual implementation files do not have to be modified. However, the customizations are limited to the ones available through properties defined in the SCA specifications that map the properties to the implementation artifacts through mechanisms such as Java annotations. The

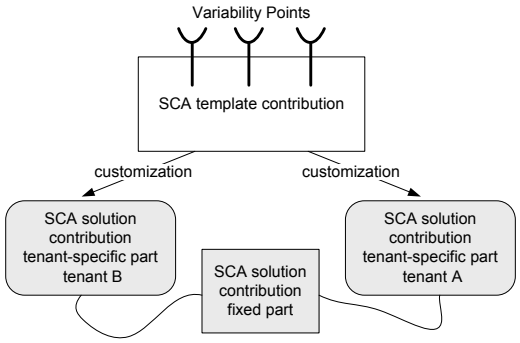

Figure 2. Templates and derived fixed and tenant-specific parts of a solution.

variability descriptor file is added to the root of the SCA deployment package. The locators inside the variability descriptor therefore can point to the artifacts included in the package by employing relative paths to the files. A configuration tool can then be used to transform the artifacts contained in the template that have open variability points (described in the variability descriptor) into configured artifacts. Configured artifacts are files where all variability points have been bound. We define an SCA solution contribution as an SCA contribution that is derived from an SCA template contribution by binding all variability points. As shown in Figure 2 for each customization derived from an SCA template contribution a separate SCA solution contribution exist that contains all the necessary artifacts as configured by the customer. This SCA solution can be deployed in any SCA-compliant middleware. In future work we will show how this SCA solution package can be used to derive

provisioning scripts to provision the application in a non-SCA environment.

3.2. Multi-Tenancy Patterns and SCA

As explained in the introduction a multi-tenant aware SaaS application is made up of two general parts. The part that is fixed for all tenants and needs to be deployed only once per SaaS application and the part that is tenant specific and needs to be deployed for each tenant. So far we have only created a solution contribution that contains both the fixed as well as the tenant-specific parts. A straightforward approach using SCA would be to define two SCA solution contributions. One contribution containing all the fixed parts and one contribution containing all the tenant specific parts. The fixed part can then be installed once per SCA domain while the tenant-specific parts are installed once per tenant and reference the fixed parts for the components needed to run the tenant specific parts. In order to decide which parts of an SCA template contribution will end up in the fixed SCA solution contribution and which parts will be contained in the tenant-specific solution contribution we introduce multi-tenancy patterns describing how multi-tenancy is achieved with a component.

Single Instance Multi-Tenant Components. The single-instance multi-tenant component pattern describes a component that is only deployed once for all tenants. All tenants use the same implementation configured through the same component. In particular this means that for example the properties, services and references configured for the component are the same for all tenants. In short: one instance of a configured implementation (a component) serves all tenants of that SaaS application. This means that the component description as well as the implementation code is contained in the composite descriptor of the fixed SCA solution contribution as shown for components A and C in Figure 6. An example of a component deployed only once is a service that performs currency conversions. It is not needed to configure the service on a per-tenant basis. Figure 3 shows the icon for this pattern.

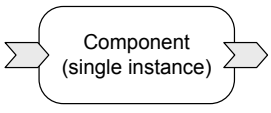

Figure 3. Single instance component icon.

Single Configurable Instance Multi-Tenant Components. The single-configurable instance multi-tenant component pattern describes a component which is deployed once for all tenants of the SaaS application that use it. However, in comparison to the single-instance multi-tenant component pattern, tenants can configure the component. In SCA terms this means that all tenants use the same implementation but each tenant deploys a separate component that references this implementation. Tenants can therefore set the properties, as well as the referenced and offered services of the implementation on a per-tenant basis. Therefore the component description is contained in the composite descriptor of the tenant-specific SCA solution contribution where the implementation is contained in the fixed SCA solution contribution as shown for component B in Figure 6. An example for a component deployed following this pattern is a service that retrieves data from a database where a tenant can configure the service to use a tenant-specific database so that other tenants cannot access or read the data of this tenant. Figure 4 shows the icon for this pattern.

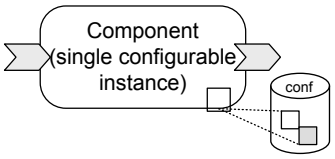

Figure 4. Single configurable instance component icon.

Multiple Instances Multi-Tenant Components. The multiple instances multi-tenant component pattern describes a component which is deployed separately for each tenant of the SaaS application that uses it. In SCA terms that means that the tenant-specific solution contribution contains the implementation as well as the description of the component in its composite descriptor as shown for component D in Figure 6. In comparison to the single-configured instance pattern the multiple instances pattern allows configurations of the component that exceed those made in the single-configured instance pattern. It is not only possible to configure the implementation of the component through the means of the composite descriptor (i.e., referenced and offered services as well as properties) but also to deploy separate implementation files, such as BPEL scripts or deployment descriptors of a Java implementation on a per-tenant basis. Figure 5 shows the icon for this pattern.

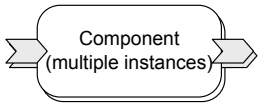

Figure 5. Multiple instances component icon.

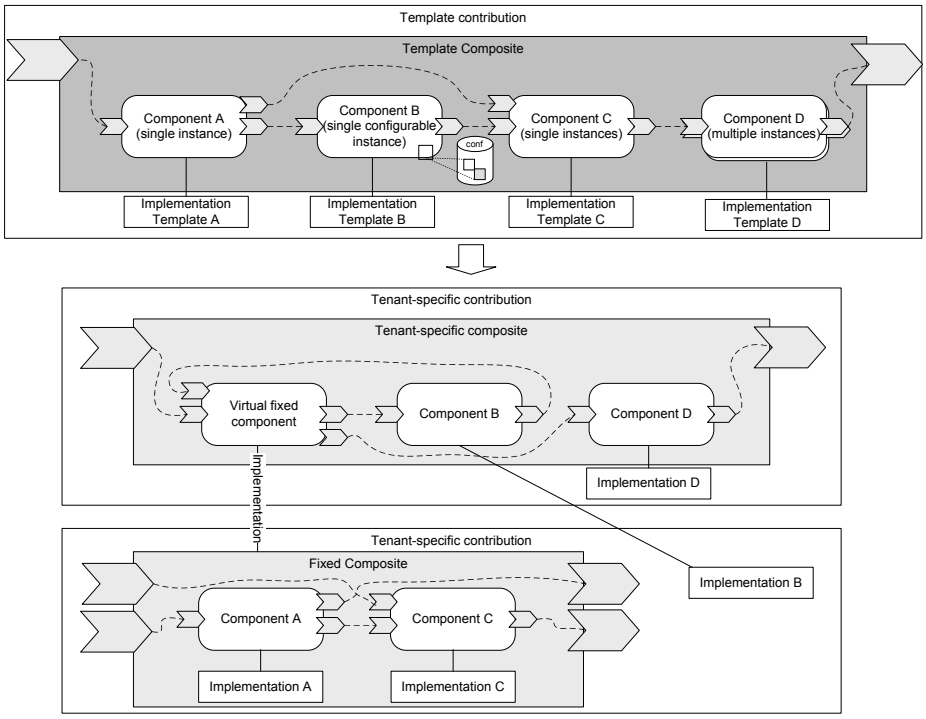

Figure 6. Transformation of template contribution into tenant-specific and fixed contributions.

Optional Components. Optional components are components contained in the SCA template contribution that can be de-selected on a per-tenant basis. For example a SaaS application contains a payment component but individual tenants choose to deselect the SaaS application's payment component because they want to include their own. The handling of optional components depends on the multi-tenancy pattern of this component. If it is deployed after the single instance pattern those tenants that do not need it simply do not reference it from other components or composites.

In case the optional component is deployed following the single-configured instance pattern the tenants that do not need the component simply do not add the component in the composite descriptor of their tenant-specific solution contribution. In the third case where the optional component is deployed using the multiple instances pattern those tenants that do not need the component do not add the implementation code and the description of the component to their tenant-specific composite descriptor and contribution package. Figure 7 shows the icon for this pattern.

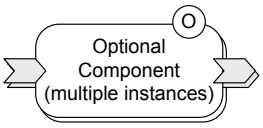

Figure 7. Optional component icon.

Tenant-Specific Components. Sometimes tenants might want to include their own components that are not specified in the SCA template contribution. In order to do so, two solutions are possible. Either the component is already deployed somewhere else, then it can be referenced via SCA specific means, where the composite promotes the services and references to the outside that communicate with the external component. The second possibility is to include the component implementation in the tenant-specific solution contribution. This can be done by including the component implementation as well as the component descriptor in the tenant-specific solution contribution.

3.3. Wiring, Services and References

Whether a component is included in the fixed solution contribution or the tenant-specific solution contribution also determines how this component is wired to other components. Components in the same composite (and therefore in the same contribution) can be wired using SCA wires as defined in the template contribution package. In case all components are deployed using the single instance pattern, the fixed solution contribution can be deployed without changing the wires between the components. The same

holds true if all components are deployed in the multiple-instances component pattern or the single configurable instances pattern as all component declarations are contained in one contribution. However, since components deployed with the different patterns can be arbitrarily mixed, situations can arise where one component declaration is located in the fixed contribution and another one is located in the tenant-specific contribution. In this case the components of the fixed part are all grouped in one composite the "fixed composite" inside the fixed contribution. All services of components that must be reached from the tenant-specific part of the application are promoted as services of the fixed composite. Similarly the references pointing to components inside the tenant-specific parts are promoted as references of the fixed composite. The fixed contribution contains an export element which exports the contents of the contribution so that they can be imported in other contributions such as the tenant-specific contribution. The tenant-specific parts of the application are placed in another contribution which contains an import element to import the contents of the fixed contribution. An additional component is added to the tenant-specific specific composite that references the fixed composite. We call the new component the "virtual fixed component" as it represents all the fixed parts of the application. The fixed composite of the fixed contribution is used as a component implementation of the virtual fixed component. Figure 6 shows how the single instance components of the template are copied into a fixed composite that is then used as an implementation of the virtual fixed component in the tenant-specific composite.

3.3.1 Multi-Tenancy Patterns and Variability

In the example in Figure 6 the multi-tenancy patterns of the components are already fixed in the template. However, these multi-tenancy patterns might also be targeted by variability points. In particular this means that users can decide which components they want to be deployed following which pattern. For example a customer might want to deploy a component in a multiple services fashion instead of the single-configurable component pattern because regulatory or performance reasons dictate that this component cannot be shared with other tenants.

4 Conclusions and Future Work

In this paper we presented a package format for service oriented SaaS applications based on the service component architecture. We described how variability descriptors can be used to annotate SCA contributions with variability points in order to describe configurable SCA application templates. The paper explained how the components in an SaaS application template can be annotated with multi-tenancy information. In particular we introduced multi-tenancy patterns as a means to specify how individual components are deployed with regard to multi-tenancy. Based on these annotations deployable contributions can be generated. Two different kinds of deployable contributions exist per SaaS application. The fixed contribution is the set of artifacts (implementations, components, etc.) that are the same for all tenants. The second part of the application is the tenant-specific package. These parts are customized for each tenant and therefore need to be deployed for each tenant separately. Once the fixed part of the application has been deployed, in order to add a new tenant only the tenant-specific parts need to be deployed. In future work we will investigate how both solution contributions can be deployed in non SCA environments by deriving deployment scripts out of the SCA solution package.

References

[1] J. Bosch. *Design and use of software architectures: adopting and evolving a product-line approach*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[2] F. Chong and G. Carraro. *Building Distributed Applications Architecture Strategies for Catching the Long Tail*. MSDN architecture center, http://msdn2.microsoft.com/en-us/library/aa479069.aspx, 2006.

[3] M. Jaring and J. Bosch. Architecting product diversification - formalizing variability dependencies in software product family engineering. In *QSIC '04: Proceedings of the Quality Software, Fourth International Conference*, pages 154–161, Washington, DC, USA, 2004. IEEE Computer Society.

[4] R. Mietzner. Using Variability Descriptors to Describe Customizable SaaS Application Templates. Technical Report Computer Science 2008/01, University of Stuttgart,Germany, January 2008.

[5] 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.

[6] Organization for the Advancement of Structured Information Standards (OASIS). Web Services Business Process Execution Language Version 2.0 Standard. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.pdf, 2007.

[7] Salesforce.com, Inc. Salesforce CRM. http://www.salesforce.com.

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

All hyperlinks in this document have been last checked on March 12th 2008