Institute of Architecture of Application Systems

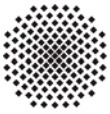
Combining horizontal and vertical composition of services

Ralph Mietzner, Christoph Fehling, Dimka Karastoyanova, Frank Leymann

Institute of Architecture of Application Systems,
University of Stuttgart, Germany
{mietzner, fehling, karastoyanova, leymann}@iaas.uni-stuttgart.de

BibTeX:

```
@inproceedings{MietznerFKL10,
  title      = {{Combining horizontal and vertical composition of services}},
  author     = {Ralph Mietzner and Christoph Fehling and Dimka Karastoyanova
                and Frank Leymann},
  booktitle  = {Proceedings of the IEEE International Conference on Service-
                Oriented Computing and Applications, SOCA 2010},
  year       = {2010},
  doi        = {10.1109/SOCA.2010.5707142  }
  publisher  = {IEEE Computer Society}
}
```

Universität Stuttgart
Germany

Combining Horizontal and Vertical Composition of Services

Ralph Mietzner, Christoph Fehling, Dimka Karastoyanova, Frank Leymann
Institute of Architecture of Application Systems
University of Stuttgart
Universitaetsstr. 38, 70569 Stuttgart, Germany
firstname.lastname@iaas.uni-stuttgart.de

***Abstract*—Service composition is a well-established field of research in the service community. Services are commonly regarded as black boxes with well-defined interfaces that can be recursively aggregated into new services. The black-box nature of services does not only include the service implementation but also the middleware and hardware to run the services. Thus, service composition techniques are typically limited to choosing between a set of available services. In this paper we keep the black-box nature and the principle of information hiding for the service implementation, but break up services *vertically*. By introducing *vertical service composition*, we allow services to be provisioned on the right middleware when they are requested, thus making service-binding more powerful as services with the desired quality of service can be provisioned *on demand*. We introduce the concept of vertical service composition and present an extension to an enterprise service bus that implements the concept of vertical service composition by combining concepts from provisioning with those of (dynamic) service binding.**

Keywords-SOA; composition; provisioning; enterprise service bus; cloud

I. Introduction

One important field of research in the service community is the field of *service composition* [14], [5]. An important aspect of service composition is the *finding* and *binding* of services in order to compose them into a *composite* application. In the traditional SOA-triangle [14], *service providers publish* their services in a *service registry* where they can be *found* by *service requestors* based on their functional and non-functional properties. Having found a suitable service, a service requestor can *bind* against that concrete service and can use it. Depending on the time in the development process of an application when concrete services are bound the binding strategy is either called *static binding* or *dynamic binding*. Static binding of services to service requestors occurs during the development of a service composition. To improve the flexibility of a composition dynamic binding is performed at deployment time or even at runtime and thus services can be exchanged without re-compiling the composition.

From the point of view of a service requestor, a service is treated as a *black box*. In particular that means that the principle of *information hiding* applies and that the requestor is only aware of the interface of the service and not of its implementation. Treating a service as a black box also means that the middleware and hardware infrastructure needed to run the service, is typically not known to the service requestor.

Another important property of a service from the point of view of a requestor is its *always on* semantics, meaning that the requestor can bind against the service at any time and does not need to instantiate or release it prior or after its usage. When binding against a service, a service requestor sends a list of functional and non-functional properties to the *selection facility* (also called repository, the bus, or enterprise service bus [4]) which returns a list of *candidate services* that match the requested functional and non-functional properties. Then the requestor selects one of the candidate services and thus binds against this service, which the requestor can then use by sending messages to this service. In enterprise service buses often a *virtual service* is used against which the service requestor can bind [4] (cf. the composite service which acts as requestor in Figure 1). In this case the requestor binds against this virtual service and delegates the dynamic binding to the enterprise service bus.

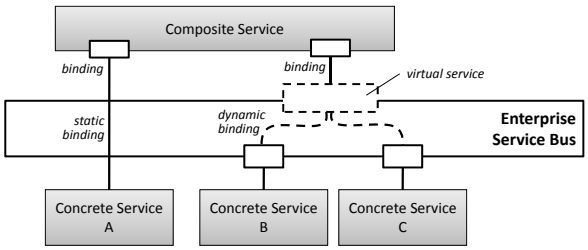


Figure 1. Service Binding in the ESB

In case the selection facility or the bus does not return any suitable candidate service, the requestor cannot bind against any service, as none of the available services fullfils the required functional and non-functional properties. One solution to this problem would be to offer services in a variety of configurations offering different non-functional properties. The drawback of this is that providers must run these services on probably different infrastructures in all possible configurations without knowing beforehand if they will ever be used by customers, thus creating bad utilization rates of the underlying data centers.

With the advent of Cloud infrastructures and "as a service" models such as infrastructure as a service (IaaS), platform as a service (PaaS) and software as a service (SaaS), *elastic* infrastructures become widely available. When combining these elastic infrastructures allowing the *on demand* provisioning and deprovisioning of services, with the concept of dynamic binding the limitation presented above can be removed. Thus, selection facilities do not only return actually running services but also services that can be provisioned on demand. Only when a requestor requests one of these services this service is then automatically provisioned with the required qualities of service such as availability, location or security.

To be able to realize this vision services must be decomposed *vertically*. Vertically decomposing a service means that its implementation is still treated as a black box. However, the underlying middleware and hardware is made explicit in form of requirements on the required middleware and hardware. The resulting effects of these requirements on the overall functional and non-functional properties of the service are also made explicit. In this paper we introduce a model that allows to decompose services vertically.

In addition, we introduce *dynamic vertical binding* as a concept allowing providers to offer *virtual components* that are provisioned and configured on demand only when they are needed, in the exact configuration that they are needed by a requestor. Thus we show how automatic provisioning and the elasticity of cloud platforms can be combined with the dynamic binding concept of service-orientation. We show, how traditional ESBs can be extended with a *vertical composition component*, that performs dynamic vertical binding in cases the traditional horizontal dynamic binding of components in the bus is not powerful enough. Thus the contribution of this paper is the following:

- The conceptual foundations for *vertical composition* (Section II).
- The *combination* of vertical and horizontal composition into dynamic vertical binding (Section III).
- An *architecture, implementation and evaluation* of a middleware - the vertical composition enabled enterprise bus (VC-Bus) that implements the concepts of vertical service composition (Sections IV and V).

II. Vertical Composition - Foundations

A. Components - Terminology

In this paper we will use the term *component* to describe clearly separated pieces of functionality in an application. A component can require other components to run by having (a) a deployment relationship on another component, i.e. the component must be deployed on another component to be able to run, and/or (b) a dependency on one or more other components, i.e. the component must use the other components to be able to perform its functionality. See [16] for a complete definition of our component model. In this model a component can be realized either by a piece of code supplied with an application or through a service provided by an (external) provider. Thus a service is a special kind of component.

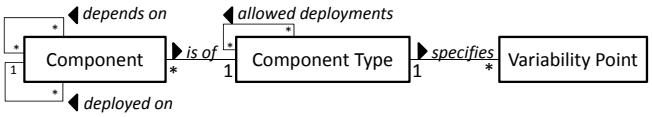


Figure 2. Components, Types and Relations

As shown in Figure 2, a component is characterized by its *component type*. Component types describe classes of components. A component type specifies so-called *variability points* that can be customized by a requester (such as functional or non-functional properties) and that influence which component of that type in the bus is selected or how components are customized prior to deployment. Other variability points of a component type, such as, for example, an EPR, or its physical location, are customized during the provisioning of the concrete component. A component type also specifies *allowed deployment* relations that specify on which other component types this particular component type can be deployed and executed. In the following if we talk about an "XY component" we implicitly mean a "component of type XY".

The so-called *host components* are components that allow other components to be deployed on them. The component type for a host component describes which other component types can be deployed on components of that type.

Components can be in any level of the application stack. Components will be symbolized by rounded rectangles in all figures of this paper. For instance, all rounded rectangles in the application stacks in Figure 3 are components of different types that are deployed on other components, i.e. a BPEL component is deployed on a BPEL engine component which is deployed on an application server component which is deployed on a virtual machine.

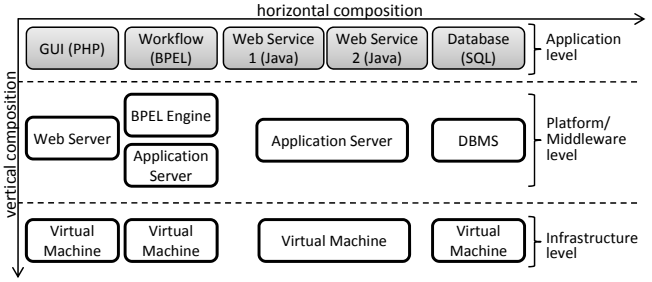


Figure 3. Components in an example application

Given our definition of components above, services in a service-based application can be seen as a special form of a

component. Thus service composition known from SOA can be seen as a special form of *component composition*. From a deployment perspective this composition is a *horizontal composition* as all services are on the same level, the application level in the application stack (cf. Figure 3).

Taking the whole hardware, middleware and software stack of a service into account, as shown in Figure 3, such a service can be seen as a composition of the required hardware, middleware and software components needed to provide the service. We call this type of composition a *vertical composition* from now on, as this corresponds to the "deployment" relationship that the individual components (software, middleware, hardware) can have on each other [12].

B. Vertical Component Composition

As shown in Figure 3, a typical component on the application level is deployed on another component which may in turn be deployed on yet another component thus these components form a vertical composition as opposed to a horizontal composition of the components at the application level.

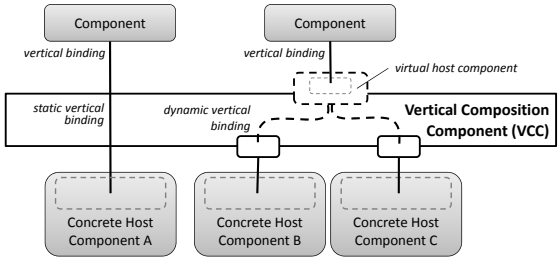


Figure 4. Vertical binding

To facilitate the understanding of the following sections we introduce the following terminology related to vertical compositions: *Provisioning* is the configuration and installation of one component on top of another component, such as the installation of a Web service component on an application server (which can also be clustered).

A *virtual host component* is an abstract component on which other components can be deployed (cf. Figure 4). The semantics behind this is that the concrete host component on which a component is to be provisioned later does not need to be known prior to the provisioning time. Software developers can therefore provision their components on these virtual host components, without knowing in advance their concrete realizations, for example, at a provider. This is especially important in settings where multiple concrete realizations of a virtual host component with possibly different quality of services might exist.

A *virtual component* is a component that is deployed on a virtual host component (see Figure 4). A virtual component cannot be used by an application unless it is bound to a concrete host component. Once a component developer deploys a component on a virtual host component it becomes a virtual component.

A *concrete host component* is a host component that is offered by a component provider and that is already running. A *concrete component* is a component that is deployed on a concrete host component.

Similar to the binding strategies in horizontal compositions we now define binding strategies for the vertical binding of components.

Static vertical binding is similar to static binding in service compositions. A component of an application is provisioned on a concrete host component. This is implicitly done in a lot of cases today. Take the example of a BPEL process component developer: He deploys his BPEL process on a BPEL engine and thus statically binds his service (implemented as a BPEL process) to a concrete instance of a BPEL engine.

Dynamic vertical binding is similar to dynamic binding in service compositions. Instead of provisioning a component on a concrete host component, the component is provisioned on a virtual host component. This means that it is not ready to be used because the virtual host component does not have an associated implementation and thus the component to be provisioned is not available yet. The semantics are again similar to a virtual service that cannot process a request as it has no implementation behind it. Thus, in case a requester requires an instance of a component that is provisioned on a virtual host component, the virtual host component must be substituted by a concrete component on which the component is then provisioned. The selection of a concrete component depends on additional criteria (such as non-functional properties) that the requester must submit to the binding middleware.

We call the component in the bus that allows the static and dynamic vertical binding as well as the definition of virtual host components the *Vertical Composition Component* (VCC in short).

C. Motivating Example

Consider the following scenario: A component developer develops a risk assessor BPEL process that orchestrates a set of Web services into a risk assessor Web service. This Web service can be used to evaluate the risk for lending a certain amount of money. Once this developer has built the BPEL process he packages it and deploys it on a concrete BPEL engine which exposes it as Web Service. The Risk Assessor process works as follows: It first detects whether the amount to be lent is above a certain threshold, if yes it invokes several Web services of credit rating companies to compute the risk. If no, it returns a low risk value without invoking the credit rating companies. Several customers have different functional requirements on the risk assessor BPEL process, one customer wants to have the threshold below 300 Euro

another one wants the threshold below 500 Euro. Additionally, they can have different non-functional requirements such as high-availability vs. no guaranteed availability and a lower price. Thus the component developer does not know on which middleware (high-available or not) and with which customization the BPEL process must later be deployed. To overcome this limitation the component developer models the BPEL process as a virtual component that must be deployed on a virtual host component (a BPEL engine) and has variability points [10] that, depending on the requested customization of the functional and non-functional properties customizes the BPEL process and influences the binding of the virtual host component to a concrete host component (namely a high-available BPEL engine or a BPEL engine with no guaranteed availability). In Section V we show how the tools we built can be used to implement this example.

III. Dynamic Vertical Binding

In this section we describe the characteristics of the Vertical Composition Component that enable dynamic vertical binding. The architecture of the VCC must contain components that support these characteristics and thus perform the needed functions.

A. Standardized Component Interfaces

One assumption behind the concept of dynamic binding is that the concrete services implementing a virtual service all implement the same interface so that the service client does not need to be changed when a different service is bound dynamically.

To be able to dynamically bind virtual components to different concrete host components these host components must also offer a unified interface so that the Virtual Composition Component knows which operation it must call. These are the interfaces used by the bus to enable the provisioning of and binding to the concrete component. In [11] we introduce the concept of *resource processes* as a concept to unify the interfaces of (host) components of the same type that are provisioned using different provisioning engines. This concept, EAI techniques or semantic mediation can be used to unify the interfaces of different infrastructures that can provision components on concrete components.

The following example illustrates the need for unified interfaces: A provider offers a virtual host component of type "BPEL Engine". Concrete realizations of this virtual host component are, for example, a clustered BPEL Engine environment at the provider managed with an IBM Tivoli Provisioning Manager provisioning engine and a normal BPEL Engine installation on Amazon's Elastic Compute Cloud (EC2) managed by Amazon's provisioning infrastructure. The BPEL engine virtual host component offers an operation which can be used by component developers to deploy a BPEL process component on that virtual host component. Deploying thus means to upload the BPEL process to the bus and specify that it must be deployed on a concrete host component of type "BPEL Engine" to be used.

Once a client application wants to use the BPEL process and specifies his service-level requirements (e.g. high-availability required or not) the BPEL process is provisioned on one of the concrete components (if high-availability is required, the clustered BPEL engine component is used).

In our implementation of the VCC, resource processes (also called component flows) are implemented as WS-BPEL processes that provide a uniform interface to the outside and transform the uniform messages to specific messages for the underlying provisioning engines. For example, we implemented a component flow that transforms the standardized messages into messages that call the Web service interface of Amazon EC2 and thus allows to start an Apache ODE BPEL engine machine image on EC2 via the standardized "provision" operation which is mapped to the "run-instance" operation of EC2. A BPEL process component can be deployed on this engine by simply calling the "deploy" operation of the component flow along with an EPR from which it can retrieve the code of the BPEL process component. The component flow then calls an "'upload"' Web Service that uploads the BPEL process component via SCP (Secure Copy Protocol) to the Amazon EC2 machine instance. A similar component flow has been implemented for an Apache ODE BPEL engine in the local data center that maps the standardized "provision" operation to a Web service that can execute the start script for the Apache Tomcat Servlet container that contains the BPEL engine and thus start the engine. The "deploy" operation of this component flow then downloads the BPEL process from a repository and copies it to the deployment directory of Apache ODE. Thus the same interface with the "provision" and "deploy" operations is present for both Apache ODE BPEL engines, despite the fact that one runs on EC 2 and one in the local data center.

B. Requirements for the Vertical Composition Component

To understand the requirements a vertical composition component for a bus must support for the vertical composition of components, we revisit the capabilities offered typically by a normal enterprise service bus. Typically a service bus offers the functionality to specify an interface and endpoint for the virtual service against which service requesters can then bind. This capability is needed to specify virtual services. Discoverability of services is enabled by a registry offered by the service bus. Service providers register their services with the registry. Service requesters look services up and receive a list of candidate services that match the requirements of the requester. Subsequently a service selection step is performed to identify the single concrete service realization for the requester to interact with. To select a concrete service from set of possible realizations of a virtual service a service bus offers a selection facility that can match requirements stated by the requester against capabilities provided by the

candidate services, which are typically non-functional or QoS requirements. The discovery and selection of services may be additionally constrained and refined by specifying the so-called *routes*. Routes pre-define a set of concrete services that can be mapped to a virtual services. The semantics of the routes is to limit the amount of possible realizations for a concrete service; typically this is done due to contractual regulations.

The Virtual Composition Component must provide similar and additional functions to enable vertical composition, and in particular dynamic vertical binding and provisioning:

Discovery of concrete components. The VCC must support discovery functionality for host components, components, virtual host components and virtual components. This implies that the VCC must offer a logical registry for each of the component types. In each of the registries the providers should be able to register their components along with their capabilities so that the bus can later match these capabilities against the requirements of a component requester. The capabilities are expressed as customizations of variability points for the variability points the component type offers. For example, a high-available host component of type BPEL engine has the "high-availability" variability point customized to "yes" whereas a BPEL engine host component that does not offer any guaranteed availability is registered with that variability point set to "no".

Virtual to concrete host component mapping. The VCC must offer routing and mapping functionality that allows to specify (i) which concrete components are realizations of which virtual host component and (ii) how a component that is provisioned on a virtual host component is then provisioned on a concrete host component based on the provisioning interfaces for the concrete host component. In our implementation of the VCC, providers can register their concrete host components through an EPR which specifies under which endpoint the unified interface for the concrete host component as specified in Section III-A is available and which component type is realized by the concrete host component.

Selection of concrete components. The VCC must offer a selection facility that can select concrete host components based on their capabilities for a set of requirements that are submitted to the virtual host component. These requirements are expressed as customizations of the variability points of the requested component type. For example, the variability point "high-availability" is customized with "yes" for one customer which triggers the selection or provisioning of a concrete host component with "high-availability" set to yes.

Provisioning. Provisioning of components on virtual host components is a two step process. First, a concrete host component must be selected and then the code for the virtual component must be configured and then deployed on that concrete host component. In our implementation we use WS-BPEL provisioning flows [11] that provision a component

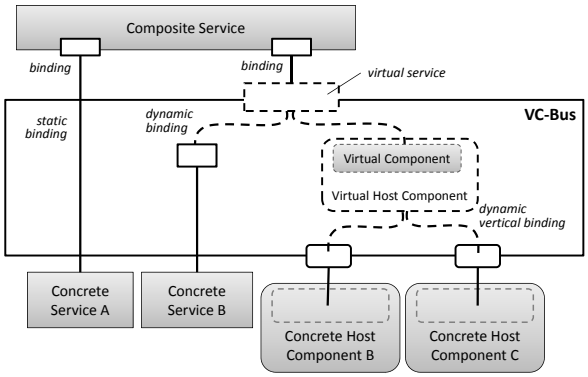


Figure 5. VC-Bus functionality

on a virtual host component by calling its standardized component interface, which is exposed as a Web service interface as described in Section III-A.

IV. The VC-Bus

In this section we describe how dynamic horizontal and dynamic vertical binding can be combined to extend a service bus with provisioning capabilities. These capabilities are used if a suitable component cannot be found during the dynamic binding and need to be provisioned first. The corresponding middleware is an extension of an enterprise service bus with the vertical composition component presented above. We call this middleware the *Vertical composition enabled bus* (VC-Bus) in short.

To specify how the two concepts of vertical and horizontal binding can be combined we need to recall what happens when a component is provisioned on a virtual host component. Once a component is deployed on a virtual host component it is not really available but a requester must issue a request so that the component is provisioned and becomes available. The result of the deployment is the following: The bus offers a virtual service that service clients can use to bind against. This virtual service is implemented by the component that is deployed on the virtual host component. Once a service client sends out a binding request to the virtual service stating the concrete requirements it has on the service (for example, in the form of a policy), the virtual service must be bound to a concrete service implementation. The bus thus searches for the available services first and in case no suitable service is found, calculates the requirements for a suitable concrete host component, that together with the component that implements the service would satisfy the requirements for the service. After the bus has found such a concrete host component the component is provisioned on that concrete host resource and the service implemented by that component is ready to answer the requests from the original service requester (see Figure 5). Thus in cases where an ESB would have rejected a request for a service because it does not have a suitable concrete service available, the vertical composition

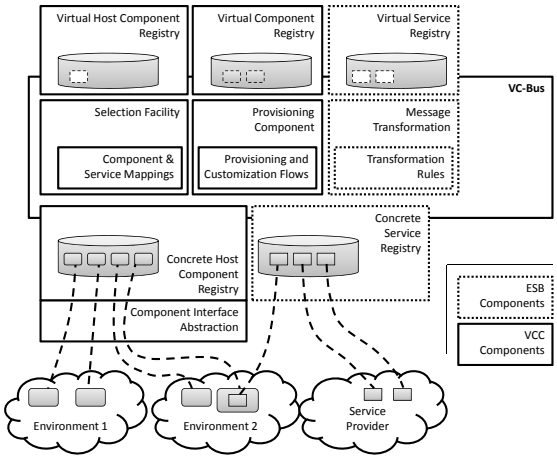


Figure 6. VC-Bus architecture

component can provision a service with suitable capabilities. Additionally the VCC can serve as a load balancer between different instances of the same service that detects when new services must be dynamically provisioned, for example, using the heuristics given in [17].

A. VC-Bus Architecture

The VC-Bus is a combination of the concepts of the enterprise service bus [4] and the concepts of the vertical composition component introduced above. Figure 6 shows the combined architecture. The VC-Bus architecture adds the VCC to the ESB. The VCC itself contains a registry for virtual host components and a registry for virtual components, as well as the provisioning component which contains the provisioning and customization flows. The component interface abstraction contains the resource processes that unify the interfaces of the underlying provisioning engines. In addition to that, a concrete host component registry is added that complements the message transformation, concrete service registry and virtual service registry commonly found in an ESB. ESBs might contain other components that are not relevant to the following chapter and thus are not included in the architecture figure.

B. VC-Bus Scenarios

The VC-Bus is not only a discovery facility as the enterprise service bus but also allows the provisioning of new components or services in case no suitable services are available to serve a request. Thus the VC-Bus can be used in scenarios where a normal ESB is not enough. The first scenario are applications with varying unpredictable user loads. This scenario is the case in software as a service or cloud scenarios where elasticity of the computing capacity is needed because the amount of tenants of an application cannot be foreseen or is changing rapidly. In this case, upon subscription of a new tenant, the capabilities of the existing service implementations for a concrete service are examined. If the capabilities are insufficient, new components that implement the services can be dynamically bound to concrete host components and added as service implementations. Once the system load for the implementations of an individual service falls under a certain threshold, a monitoring component can then use the VC-Bus functionality to remove concrete components from the environment. Another use-case where the VC-Bus has advantages over a traditional ESB is when the functionality of a component needs to be supplied for different customers with varying qualities of service and the quality of service of the components heavily depends on the underlying infrastructure. Examples are BPEL processes or Web applications where the availability and performance heavily depends on the underlying middleware. In case a service requester requests a service that is high-available but none is available, the VC-Bus can provision such a service on a high available concrete host component for the time when it is needed whereas an ESB would rely on a pre-provisioned high available service that would have been idle before the request. Thus the VC-Bus helps to reduce over-provisioning of data centers by enabling the elastic provisioning and de-provisioning of resources. This is especially useful in a cloud based environment.

C. VC-Bus Chaining

Similar to enterprise service buses, VC-buses can be chained. An example for VC-Bus chaining is when one virtual host component provided at one bus is bound to a virtual component at another or the same bus. Once a virtual component deployed on the virtual host component of VC-Bus 1 is requested its virtual host component is bound to the concrete host component that is realized by a virtual component in VC-Bus 2. VC-Bus 2 then treats this as a request to provision this virtual component on a concrete component. Once this is done the original virtual component can be deployed on the now available concrete component managed by VC-Bus 2.

V. Evaluation

To evaluate the presented architecture of the vertical composition enabled ESB we implemented several prototypes as part of the Cafe project[1]. Figure 7 shows our Eclipse-based tool support to model components and their virtual host components in the left pane as well as variability points in the right pane. The component shown in the left pane is a BPEL process that must be deployed on a virtual host component of type Apache Ode (an open source BPEL engine). On the right some of the variability points from the example in Section II-C are shown. Serializations of the shown models and the code of the risk assessor BPEL process are then packaged into a Cafe archive file (.car) which is a special kind of .zip file and uploaded via a Web interface to the

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

virtual composition component in the bus. In our prototype we realized the VCC as a separate component that is called by an ESB such as the extended Apache Service Mix ESB as presented in [13]. In general VCC can be either located on the same machine as the ESB or can be distributed over other machines, or can even be used as a standalone component. It simply must expose a set of Web service interfaces that the ESB can call. The VCC uses WS-BPEL provisioning flows and customization flows as presented in [10] and [11] to trigger the provisioning of concrete host components (in this case the Apache ODE engine) and deploy the application component (in this case the BPEL processes) on top of them. These provisioning and customization flows are automatically generated from the models contained in the uploaded .car file, details on the generation algorithms are out of the scope of this paper but can be found in [11], [12].

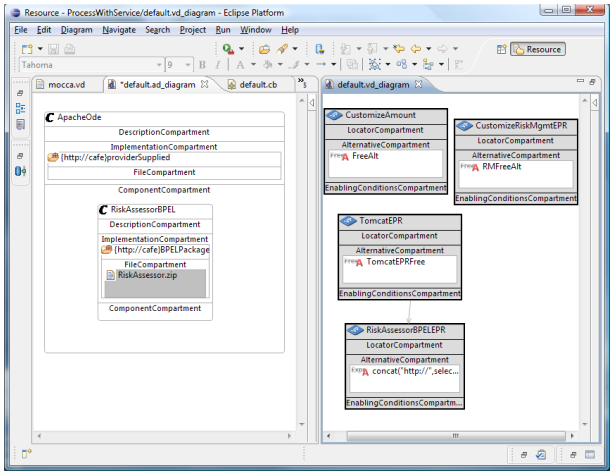


Figure 7. Service modeling tool

Table I shows the average execution time for provisioning of the Apache Ode virtual host component and the RiskAssessor BPEL process on top of it. Scenario 1 shows the time for the setup (provisioning and configuration) of the virtual host component and the component in case the virtual host component is part of an Amazon EC2 machine image and an instance of that image must first be started. Scenario 2 shows only the time of the deployment of the BPEL process component on an already started virtual host component at Amazon EC2. Scenario 3 shows the deployment of the BPEL process component on an already started BPEL engine on the local network. In general, the figures below depend heavily on the time needed to start the host component and the time to deploy a component, which are properties of the respective provisioning middleware and network and not of our approach. The figures are thus stated below only to give a general feeling on how long it takes to setup an example concrete host component and other components on top of it.

The performance figures given in Table I show that the general approach is feasible in environments where the

	Setup Time Host Component	**Setup Time Component**
Scenario 1	124 s	25 s
Scenario 2	n/a	24 s
Scenario 3	n/a	18 s

Table I
SETUP TIMES FOR APACHE ODE HOST COMPONENT AND RISK ASSESSOR BPEL COMPONENT

setup time of a component may take several minutes to complete. These are cases, for example, when a new customer subscribes to an application and thus some or all services of this application must be newly provisioned. Another case is elastic scaling when a monitoring component detects that a possible bottleneck arises and thus can provision a new instance of a service that is then used to serve future requests. However, given the setup time of several seconds it is not feasible to bind a new component to a concrete host component based on an incoming request that must be immediately handled by the newly provisioned component. Thus our approach works fine for the scenarios outlined in Section IV-B.

VI. RELATED WORK

Existing work related to the VC-Bus approach presented in this paper, can be categorized in two main categories: service composition and provisioning approaches. There exist numerous approaches in the service composition community on how to select a concrete service [5] from a set of available services. Some approaches use qualities of services as selection criteria for the selection of concrete services for a virtual service [18], [15], [13]. Other approaches use semantic technology to find suitable concrete services for a virtual service, often these approaches also allow to (semi-) automatically derive mediators that mediate between different service interfaces [3], [8]. These approaches can be reused in the VC-Bus to do the actual selection of concrete host components for a virtual host component and to do the selection of already provisioned services. However, all these approaches cannot provision new services in case they do not find a suitable one.

A pattern-based approach for the modeling of infrastructure for service oriented applications has been described in [6], [1]. This approach allows application vendors to model complex infrastructure requirements as so-called *virtual units*. These virtual units are similar to our virtual components and can be used to describe the virtual components in our vertical composition component. Virtual units in [1] can be realized by so-called *concrete units* which correspond to our concrete components. The main difference to our approach is that the approach presented in [1] focuses on the modeling and provisioning, while our approach focuses on a combination of provisioning and dynamic binding thus

combining the approaches from the systems management and service composition domain.

Similar to the VC-Bus, the Grid community also deals with the virtualization of resources, e.g. through resource brokers [7]. However, these approaches allow only for the provisioning of new components on already existing grid resources and not on virtual resources. Approaches from autonomic and cloud computing [9], [2] allow the automatic provisioning and elastic scaling of computing resources. Thus these commercial and open source cloud environments as well as available provisioning engines implement the provisioning component of the VC-Bus. However, the VC-Bus goes one step further by integrating different environments (that may come from different cloud providers) and combining the vertical composition of components offered by provisioning engines with the horizontal composition of components offered by the ESB. Additionally the VC-Bus abstracts from the concrete provisioning engines used and thus allows application developers to abstract from the concrete environment when building, deploying and reusing components for their applications.

VII. Conclusions

In this work we identify the different types of component compositions that are applied in application development. We introduce a classification that distinguishes between horizontal and vertical decomposition of applications into components. Traditional ESBs support the horizontal composition of components, however they do not support the vertical component aggregation. Therefore, a novel type of infrastructure is needed in order to support vertical component composition. In this work we specify the architecture of this infrastructure called *vertical composition component* VCC as well as its sub-components and the interactions among them. The ability of the vertical composition component to support vertical composition of components makes it a suitable infrastructure for the extension of a traditional ESB as it enables the reuse of components on all layers (software, middleware, hardware) of an application. We call the resulting middleware the *vertical composition enabled bus*, aka VC-Bus. The architecture and functioning of the VC-Bus also belong to the contributions of this paper. We presented a modeling tool, runtime and some performance figures that show the viability of the approach.

References

[1] W. Arnold, T. Eilam, M. Kalantar, A. Konstantinou, and A. Totok. Pattern Based SOA Deployment. In *Proc. ICSOC 2007*, 2007.

[2] R. Buyya, C. Yeo, S. Venugopal, J. Broberg, and I. Brandic. Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation Computer Systems*, 25(6):599–616, 2009.

[3] L. Cabral, J. Domingue, S. Galizia, A. Gugliotta, V. Tanasescu, C. Pedrinaci, and B. Norton. IRS-III: A broker for semantic web services based applications. *Lecture Notes in Computer Science*, 4273:201, 2006.

[4] D. Chappell. *Enterprise service bus*. O'Reilly Media, Inc., 2004.

[5] S. Dustdar and W. Schreiner. A survey on web services composition. *International Journal of Web and Grid Services*, 1(1):1–30, 2005.

[6] K. El Maghraoui, A. Meghranjani, T. Eilam, M. Kalantar, and A. Konstantinou. Model driven provisioning: Bridging the gap between declarative object models and procedural provisioning tools. In *Proce. ACM/IFIP/USENIX 2006*, 2006.

[7] I. Foster and C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 2004.

[8] D. Karastoyanova, B. Wetzstein, T. van Lessen, D. Wutke, J. Nitzsche, and F. Leymann. Semantic Service Bus: Architecture and Implementation of a Next Generation Middleware. In *Proc. of the 2007 ICDE Workshop*, 2007.

[9] J. Kephart and D. Chess. The vision of autonomic computing. *Computer*, pages 41–50, 2003.

[10] R. Mietzner and F. Leymann. Generation of BPEL customization processes for SaaS applications from variability descriptors. In *IEEE International Conference on Services Computing, 2008. SCC'08*, 2008.

[11] R. Mietzner and F. Leymann. Towards provisioning the cloud: On the usage of multi-granularity flows and services to realize a unified provisioning infrastructure for saas applications. In *Proc. SERVICES'08*, 2008.

[12] R. Mietzner, T. Unger, and F. Leymann. Cafe: A Generic Configurable Customizable Composite Cloud Application Framework. In *Proc. CoopIS 2009 (OTM 2009)*, 2009.

[13] R. Mietzner, T. van Lessen, A. Wiese, M. Wieland, D. Karastoyanova, and F. Leymann. Virtualizing Services and Resources with ProBus: The WS-Policy-Aware Service and Resource Bus. In *Proc. ICWS 2009*, 2009.

[14] M. Papazoglou and D. Georgakopoulos. Service-oriented computing. *Communications of the ACM*, 46(10):25–28, 2003.

[15] F. Rosenberg, P. Celikovic, A. Michlmayr, P. Leitner, and S. Dustdar. An end-to-end approach for qos-aware service composition. *Proc. EDOC*, 2009.

[16] T. Unger, R. Mietzner, and F. Leymann. Customer-defined Service Level Agreements for Composite Applications. *Enterprise Information Systems*, 3(3), 2009.

[17] B. Urgaonkar, P. Shenoy, A. Chandra, P. Goyal, and T. Wood. Agile dynamic provisioning of multi-tier internet applications. *ACM Trans. Auton. Adapt. Syst.*, 3(1):1–39, 2008.

[18] L. Zeng, B. Benatallah, A. Ngu, M. Dumas, J. Kalagnanam, and H. Chang. QoS-aware middleware for web services composition. *IEEE Trans. Softw. Eng.*, 30(5):311–327, 2004.