Constructing Web Services out of generic Component Compositions

Johann Oberleitner and Schahram Dustdar

Distributed Systems Group
Information Systems Institute
Vienna University of Technology Argentinierstrasse 8/E1841
A-1040 Wien, Austria
{joe,sd}@infosys.tuwien.ac.at

Abstract. Todays information systems are built using various component models such as Enterprise Java Beans, JavaBeans, Microsoft COM+, and CORBA distributed objects. In this paper we argue that it is crucial for designers of information systems to interactively build and test systems constructed from (a) components (enabling interoperability across component models) and (b) Web services at the same time. The contribution of this paper is threefold: Firstly, we introduce a visual tool the Component Workbench - for designing information systems out of components from different component models (e.g. EJB, COM+, CORBA) and combine them with Web services. Secondly, we show how component compositions can be turned into Web services using SOAP as a communications protocol. Thirdly, we show how to interactively test compositions before creating the actual Web services out of components.

Keywords: composition, components, component models, interactive testing

1 Introduction

Increasingly Web services [1] are gaining momentum for intra- and interorganizational integration of information systems. Current systems are built using various component models such as Enterprise Java Beans [2], JavaBeans [3], Microsoft COM+ [4], and CORBA distributed objects [5]. Furthermore, we may safely expect that required application logic will continue to be developed using those component models in the foreseeable future. Most of todays Web service development systems are based on the idea that components (and legacy systems) used in organizations should be first developed using component models and as a next step be equipped with Web services interfaces and communications means (i.e. SOAP [6]). The argument of this paper is that it is crucial for designers of information systems to interactively build and test systems constructed from (a) components (enabling interoperability across component models) and (b) Web services at the same time. In order to provide a proof-of-concept, we introduce a prototype system - the Component Workbench based on our Vienna Component Framework.

The contribution of this paper is threefold: firstly, we introduce a visual tool the Component Workbench - for designing information systems out of components from different component models (e.g. EJB, COM+, CORBA) and combine them with Web services. Secondly, we show how component compositions can be turned into Web services using SOAP as a communications protocol. Thirdly, we show how to interactively test compositions before creating the actual Web services out of components.

The remainder of the paper is structured as follows. In section 2 we explain background information about the Vienna Component Framework (VCF) we have built to access different component models in a uniform way. How component compositions can be built with VCF are illustrated in section 3. The Component Workbench, our graphical tool, is illustrated in section 4. How such compositions are turned into Web services is explained in section 5. Section 6 discusses related work. Section 7 concludes the paper and provides an brief overview for our future research directions.

2 The Vienna Component Framework

We have built the Vienna Component Framework (VCF) [7] to support the interoperability and composability of components from different component models such as Enterprise JavaBeans (EJB) [2] or Microsoft COM+ components [4].

VCF provides a Java API to reuse components that adhere to different component models within one single application. VCF abstracts the internals of the different component models, therefore simplifies the use of different component models and reduces the difficulties inherent in these models.

VCF uses plugins to simplify the extension with new component models. Currently, we have implemented plugins that provide uniform access for components that adhere to either JavaBeans, Enterprise JavaBeans, Microsoft COM+, CORBA distributed objects, and Web services that use SOAP as communication medium.

Each plugin provides the functionality to access a component model's metadata facility to find out about the *operations*, *properties*, and *event callbacks* a component supports. For each of those features VCF defines an interface that contains operations for accessing them. For instance, the interface for a components method has operations invoking this method and query methods to find out about the method parameters and its return values. The interface for callback events allows registration and unregistration of notification listeners for the events. The plugin provides implementations of each of the interfaces to provide uniform access for the corresponding component model.

Clients do not use plugin classes directly, but use a façade class to access the features of components. This allows the integration of new functionality in the façade class for all component models and all components, without changing the syntactic or semantic structure of a plugin. Furthermore, this supports per-instance modification of components on the external component interface.

A client creates an instance of a component by using a factory method that is aware of all plugin classes. This factory method takes as parameters data to identify a component's model and the corresponding plugin class and plugin-specific information for instantiating the component. The factory method returns an instance of the façade class that contains an instance of the appropriate plugin and a component-model dependent reference to the instance of the component. During this instantiation step the plugin creates instances of the implementation classes for each feature found with the metadata facility. For each operation found, an instance of the implementation class that implements the interface `IMethod` will be created and stored in a feature container (see figure 1). The same happens for properties and event-sets.

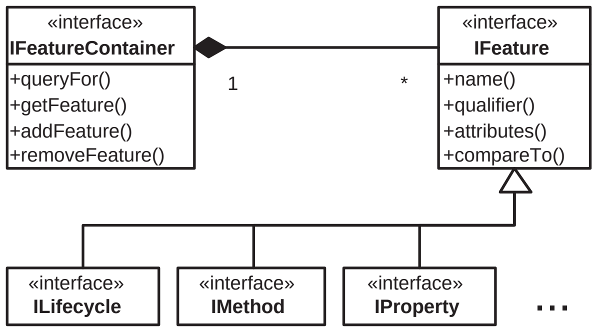


Fig. 1. Features and Feature Containers

The façade class provides a flexible mechanism to make queries for features, such as a component's operations. The queries return the feature interfaces that provide mechanisms to access a component instance's functionality.

The set of feature interfaces allows to use different components in a reflective programming style, similar to Java reflection or CORBA Dynamic Interface Invocation. This programming style is well supported by graphical composition environments such as the Component Workbench (section 4) or when components are added at runtime but is tedious to use by programmers, since even the simplest calls get complicated. Hence, we allow the generation of wrappers around the feature interfaces. This is more convenient for programmers, allows a programming style similar to use regular Java classes, and is statically typed.

Table 1 shows the component models supported by VCF, the metadata facilities used for each model, and how dynamic calls are built. Although the existing plugins all rely on such built-in metadata facilities other plugins could use data provided by component clients. This can be used to build plugins for legacy systems that do not adhere to a particular component model.

Component Model	Metadata	Dynamic Calls
COM+	Type Library	IDispatch interface
CORBA	Interface Repository	Dynamic Invocation Interface (DII)
EJB	Java Reflection	Java Reflection
JavaBeans	Java Reflection	Java Reflection
Web services (SOAP)	WSDL	Dynamic SOAP calls (WSIF)

Table 1. VCF plugins

3 Component Composition

One of the primary goals of VCF was to allow the use of components built for different component models in the same client application. VCF supports the design and implementation of such applications with one unified programming model for component models that can be accessed with a plugin.

Using Java, VCF resembles the use of JavaBeans or any regular Java classes. Furthermore, VCF supports component composition with connectors that link components with predefined communication semantics. This allowed us to support different communication primitives in different connector types such as building a connector for component method calls or component callbacks.

Similar to building component model plugins, connectors use a similar plugin architecture. Each connector plugin has to provide an implementation for the `IConnectorControl` interface. As shown in figure 2 this interface defines a method for making a connecting, and another one for canceling a connection.

In addition each connector supports a number of roles. These roles represent the end-points of a connector (see figure 3).

We do not restrict ourselves to binary connectors, but allow an arbitrary number of roles, allowing virtually any kind of connection among an arbitrary number of components. The connector plugin has to provide implementation

```
public interface IConnectorControl {

  public void connect () throws Exception;
  public void disconnect () throws Exception;
}
```

Fig. 2. Interface Declaration for IConnectorControl

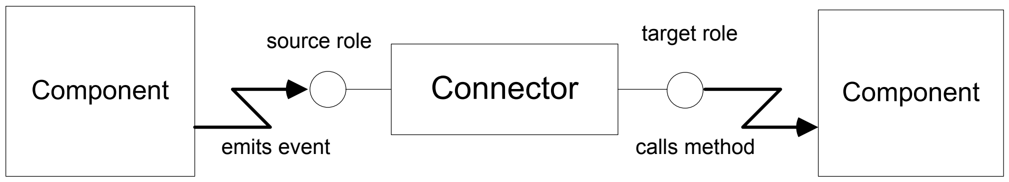


Fig. 3. Connector, Roles, and Bindings

classes for roles. Figure 4 shows the interface that has to be implemented for one particular role. It has only one method that links a connector's role, i.e. a connector's end-point, to a component binding.

```
public interface IRole{

  public void linksTo (IBinding binding) throws Exception;
}
```

Fig. 4. Interface Declaration for IRole

A binding is an encapsulation of a concrete component plus an executable part of a component. A typical example of connectors are event connectors that model component callbacks. Such callbacks are realized by VCF's event set feature. This feature allows clients to register notification listeners that implement a particular Java interface.

The source role of this connector will be linked to a particular component's event set feature while the target role will be linked to one or more method feature. The corresponding bindings contain also all arguments necessary to call the bound feature. For instance, in case of the event connector the target binding will bind a method and all argument values for calling this method. These argument values in turn can be retrieved by bindings to other components, hence allowing for recursively resolved structures.

Bindings are evaluated either at connect time of the connector, or at run-time when a certain communication between roles happens. In case of the event connector the source role binding calls the register listener method of the event set feature. This happens when the connector is made concrete, with the `IConnector Control` interface. On the target role side, a particular method is called, and arguments are forwarded. This is done only when an event occurs.

Since VCF can be enhanced with new connector types and bindings many different compositions among components are possible.

Two other VCF characteristics ease the construction of compositions. First, since no client addresses components directly but always uses a façade class for accessing a component's feature, it is easily possible to instrument this façade class for modifying the behavior of one particular component's feature. For instance, it is possible to inform other components or other clients when a particular method has been called by clients. This facility exists, even when a different component model will be used later.

Second, VCF allows the arrangement of components, connectors, or both in composite components. This allows the construction of hierarchical composition structures. Nevertheless, it is possible to expose a subset of those composition elements to clients that reside on other levels, also allowing conversational composition.

4 The Component Workbench

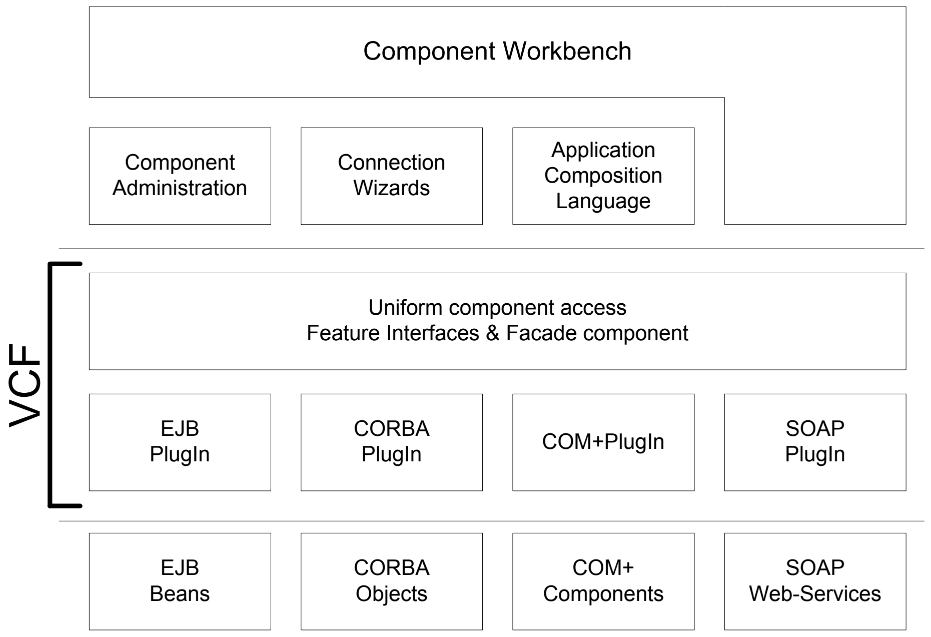


Fig. 5. Architecture of the Component Workbench

This section briefly introduces the CWB [8] which is a graphical composition environment developed in [9]. It allows developers to construct compositions out of existing components in a graphical and interactive way. Figure 5 shows the architecture of the CWB. It uses VCF to allow the use of arbitrary component models and it also supports the connector mechanism described in section 3. Component instances can be arranged within other components, so-called composite components, hence building a hierarchical structure of components. Internally a composite component consists of instances of child components that are linked by connectors. A composite component is represented either by an entire window to allow modification of its child components or it can be represented by a graphical icon in its parent component. This allows to link together composite components to other components.

Once, a component instance has been put onto a CWB window they are fully functional. This means it is possible to change a component instances' state or call an instances' operation directly within the CWB.

A wizard allows the user to define the roles and its bindings within the CWB by drawing connections. A connection can be visualized by a class provided by the connector plugin. For binary connections, by far the most common, this means that an arrow will be drawn from the source-role to the target-role. However, the visualization is not limited to this kind of representation.

A composition built with the CWB can be stored to an XML file that describes the arrangement of components and connectors [10]. Later this file can be reloaded to work on a composition again. The set of available components is stored in a different type of XML file that describes the necessary parameters to instantiate a component and the component model to which it adheres. Another type of XML file is available that describes connector types, its roles, the classes that implement the connector and the required wizard dialogs. New components or connectors can be added to the CWB by adding a new file that describes a component or a connector to the appropriate directory.

5 Web service Construction

A primary contribution of this paper is to show how a component composition built with VCF can be turned into a Web service that uses SOAP as communication protocol. This section explains how a composition that has been built with the Component Workbench can be converted into a Web service.

We propose a tool-supported approach that relies on several steps. Each of these steps corresponds to a Component Workbench wizard. Initially the developer invokes the Web-Service Designer (section 5.1) to create a new Web service. This allows for defining static properties of a service, such as its service name. The Message Designer (section 5.2) enables the definition of messages used within services. These messages can be linked to components as described in section 5.3. Figure 6 shows a composition of the components C1, C2, C3. The tasks of the Message Designer and the Component Link wizard are to let ingoing operations map to invocations of appropriate components. A service modeled with CWB can be used to create Java source code (section 5.5) and the corresponding description information. The CWB also allows interactive testing of Web services built out of components (section 5.4).

5.1 Web service Designer

The developer has to select one of CWB's composite components to be converted to a Web service or a fresh composite component can be created if the developer wants to design a new composition. The developer als can propose a name and a namespace for the service.

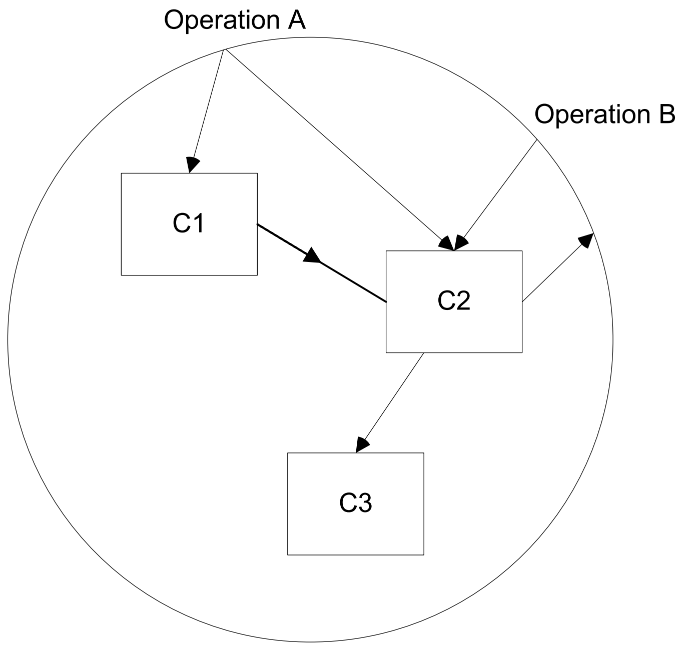


Fig. 6. Web service boundaries

5.2 Message Designer

The message designer allows users to define new messages for operations used in Web services. In the CWB this message designer is implemented as a dialog that allows the user to compose messages out of existing data types.

Since SOAP messages rely on XML Schema declarations the result of this designer is an XML Schema declaration. Furthermore, the user can provide data to fill the types declared in the wizard for both, documentation and testing purposes. Since VCF uses Java as implementation language, Java classes will be created automatically that match the newly created Schema types.

To support users in defining new messages the designer allows browsing of existing components, and Web service definitions to take SOAP *message part types* and *part names* from.

5.3 Message-Component Links

Links from components to messages and links from messages to components are defined with the Message-Component Link wizard. This wizard provides mappings from messages to methods. An ingoing message can lead to the invocation of one or more methods. Different parts of an ingoing message can be distributed to the parameters of the components' methods being called. The methods can be called synchronously or asynchronously. If one of the methods is called synchronously, the whole Web service supports synchronous communication semantics. Otherwise the Web service supports asynchronous interaction semantics. Return messages are mapped similar to the ingoing messages.

When the user has designed messages that use *part types* and *part names* that identically map to types and names of components then the message-component link wizard automatically proposes standard mappings.

Furthermore, it is possible to add filters on ingoing and outgoing messages for parameter conversion. The actions that are invoked when a message occurs are shown visually.

Figure 7 shows a simplified representation of the wizard. On the left an input message for a student grading web service is displayed. Each row shows the parts of this message with its part type and part name. On the right part component invocation boxes are shown. These invocations are initiated when the input message arrives. The figure shows that the parts of the message have a different natural order than the arguments of the gradeDB's `storeGrade` method. Hence, the wizard shows this reordering with arrows that start in the parts and end in the arguments. Furthermore, the input message's instructor part does not match the type of the instructor argument of the `storeGrade` method. Hence, an additional call that makes a lookup for an id of the instructor in an instructor database has been added. This call is shown as another invocation box. The box has an arrow on its right side that leads to the instruction parameter of the storeGrade method.

5.4 Interactive Testing

When a Web service has been declared, and its messages and the links to components completed this composition can already be tested within the CWB. It is possible to open another window, instantiate the designed Web service, connect CWB provided components that allow entering of ingoing message parts, and displaying return value parts. Although the Web service is not created at this time, it is possible to test the composition as if it were a Web service. In addition, later when the Web service has been created it is possible to use the

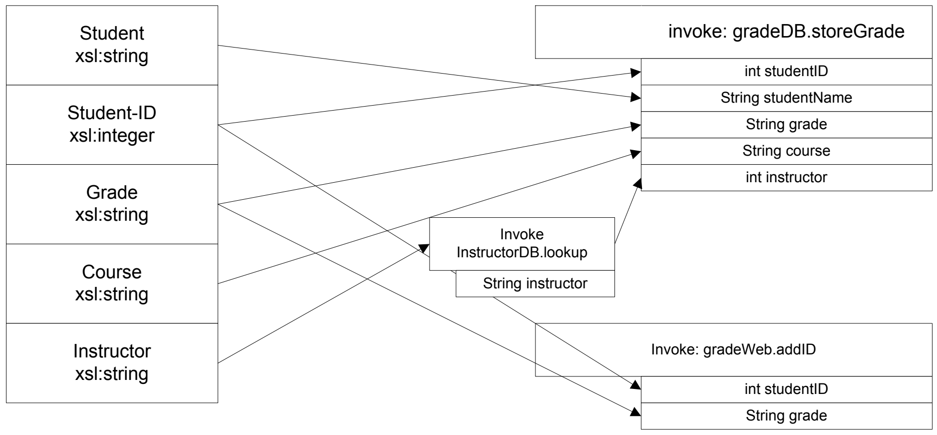
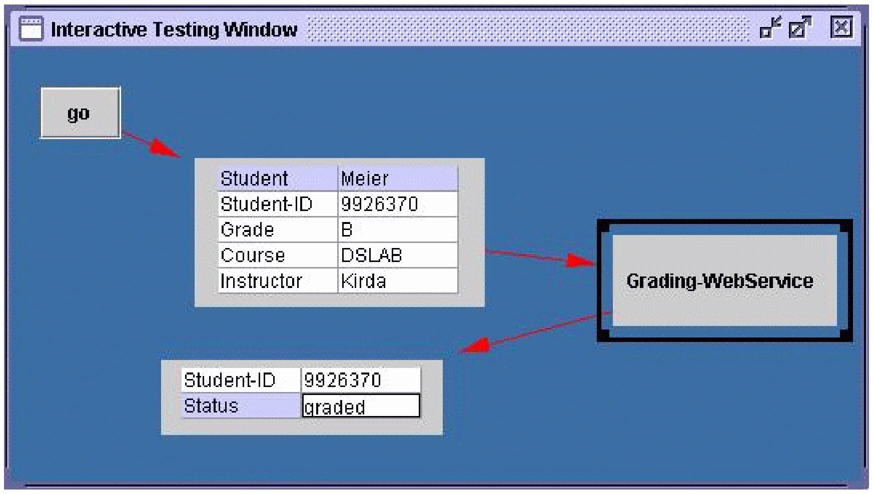


Fig. 7. Message Component Link Wizard

same test case. Instead of the designed Web service, the concrete Web service is used.

The CWB provides components to enter and display message arguments, directly on the screen. These components use VCF to find out about the method parameters, and provides an input mask for entering and displaying message parts.



Student	Meier
Student-ID	9926370
Grade	B
Course	DSLAB
Instructor	Kirda

Student-ID	9926370
Status	graded

Fig. 8. Interactive testing

Figure 8 shows how a *student grading Web service* is connected to an input component that allows entering the part values of the message and an output component that displays the acknowledgment message. The communication is started when the user presses the *ok* button.

5.5 Exporting the Web service

Once the messages, and the links to the corresponding components has been defined, the source code for a Web service can be generated. We currently support the generation of source code that uses the Apache SOAP engine. For each Web service operation, a method is generated that uses VCF to call the corresponding components. The Component Workbench uses an XML format for storing component compositions [10]. We apply an XSLT style sheet to convert such a composition into the source code for the Java Web service. We plan to support different target environments in the future.

In addition to the generation of the source code the Web service description is created. The information gained from the user in the Web service designer, and the message designer is used to create this file.

6 Related Work

Recently, initiatives have been started to provide transparent SOAP access for Enterprise JavaBeans [11] and CORBA objects [12]. Similarly, Microsoft supports the access of COM+ objects via SOAP in a recent release of the Internet Information Server (IIS). However, most of these initiatives are currently only specifications and implementations are rare.

Graphical composition environments have gained some attention by the research community in the last years [13]. The CWB is specific with respect to the support of arbitrary component models. VCF supports interoperability across component models. Microsoft's .NET framework supports transparent kind of functionality for COM+ components, .NET classes, and SOAP Web services [14]. However, support for applications that rely on Sun's JDK cannot transparently integrated into .NET easily. IBM's Web service invocation framework (WSIF) [15] has similarities to VCF. Like VCF it is primarily used on the client, and provides transparent access to different providers such as SOAP Web services. A recent addition was a provider that supports Enterprise JavaBeans.

Some commercial application servers that support Web services provide tools to define workflow applications. These tools can be compared to our extensions of the Component Workbench for exporting component compositions. Usually these tools are restricted to Web services itself and don't support the use of multiple component models.

7 Conclusions and Future Work

This paper presented a graphical composition environment for integrating arbitrary component models as well as Web services and enabling interoperability across them. Today, most tools in the Web services domain are focused (and restricted) solely to composition of Web services. In our paper we demonstrated a prototype system allowing the use of a combination of multiple component models and Web services within one system. We think that such an open and integrated approach will foster the Service-oriented computing approach and demonstrate its viability. Our future work includes increased attention towards interactive testing of Web services, which becomes more and more challenging, while Web services providers are distributed on the Internet and Quality-of-Service issues become more relevant [16].

References

1. Benatallah, B., Casati, F., Toumani, F., Hamadi, R.: Conceptual modeling of web service conversations. In: Proceedings of 15th International Conference on Advanced Information Systems Engineering (CAiSE'03), Springer (2003)
2. DeMichiel, L.G., Yal cinalp, L.Ü., Krishnan, S.: Enterprise JavaBeans Specification, Version 2.0. Sun Microsystems. (2001)
3. Hamilton, G., ed.: JavaBeans. Sun Microsystems, http://java.sun.com/beans/ (1997)
4. Kirtland, M.: Designing Component-Based Applications. Microsoft Press (1999)
5. Siegel, J.: CORBA 3: Fundamentals and Programming. Second edn. John Wiley & Sons, Inc. (2000)
6. W3C: SOAP - Simple Object Access Protocol. (2001)
7. Oberleitner, J., Gschwind, T., Jazayeri, M.: The Vienna Component Framework: Enabling composition across component models. In: Proceedings of the 25th International Conference on Software Engineering (ICSE), IEEE Press (2003)
8. Oberleitner, J., Gschwind, T.: Component distributed components with the component workbench. In: Proceedings of the 3rd International Workshop on Software Engineering in Middleware 2002 (SEM), LNCS 2596, Springer (2002)
9. Oberleitner, J.: The Component Workbench: A Flexible Component Composition Environment. Master's thesis, Technische Universität Wien (2001)
10. Oberleitner, J., Gschwind, T.: Transforming application compositions with xslts. In Assmann, U., Pulvermueller, E., Borne, I., Bouraqadi, N., Cointe, P., eds.: Electronic Notes in Theoretical Computer Science. Volume 82., Elsevier Science Publishers (2003)
11. Sun Microsystems: Enterprise JavaBeans Specification, Version 2.1 - proposed final draft. (2002)
12. OMG: CORBA-WSDL/SOAP Interworking. (2003)
13. Lüer, C., van der Hoek, A.: Composition environments for deployable software components. Technical Report UCI-ICS-02-18, Department of Information and Computer Science, University of California, Irvine (2002)
14. Richter, J.: Applied Microsoft .NET Framework Programming. Microsoft Press (2002)
15. Duftler, M.J., Mukhi, N.K., Slominski, A., Weerawarana, S.: Web Services Invocation Framework (WSIF), http://ws.apache.org/wsif/references.html. (2001)
16. Zeng, L., Benatallah, B., Dumas, M., Kalagnanam, J., Sheng, Q.Z.: Quality driven web services composition. In: Proceedings of the 12th International World Wide Web Conference 2003 (WWW), ACM (2003)