Workflow-Based Composition and Testing of Combined e-services and components

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. Information systems are increasingly built using component models such as Enterprise Java Beans, COM+, or CORBA objects. Dynamically aggregated and composed Web services can be seen as a newly emerging research area for Service oriented architectures, which are combined with components. Our paper presents three main arguments: firstly, that composition of combined e-services and components is increasingly gaining momentum; secondly, that those compositions should be workflow-supported by workflow-based languages such as BPEL4WS, and thirdly, that (semi)automatic and interactive testing of those combined services and components is of paramount importance. This paper presents the underlying framework as well as an implementation of a workflow-based composition system.

Keywords: Web services, Workflows, Coordination, Orchestration, Middleware

1 Introduction

Service oriented Computing (SOC) is a promising research area which is receiving considerable attention. This evolution in distributed computing builds on many aspects and insights gained from object-oriented and component computing. The SOC model puts services and their composition, interaction, and orchestration into the forefront of concerns. Today information systems are increasingly built using component models such as EJB, COM+, and CORBA objects. The aim is to ultimately provide services in the sense of autonomous platform-independent computational elements that can be described, published, discovered, orchestrated and programmed. Composition and enactment of service workflows face a number of challenging issues that need to be addressed. The challenges discussed in this paper include:

- How can Web services be tested (semi)automatically before deployment and runtime?
- How can abstract Web services Workflows be represented visually?

- How can one ensure efficiency (in terms of performance) of composed Web services Workflows?
- What are suitable techniques for executing composed process (i.e. workflows)?
- How can one efficiently use monitoring techniques for run time analysis of Web services process executions?

The contribution of this paper is as follows: (i) methods for *combined* composition of e-services and components is increasingly gaining momentum; (ii), those compositions should be workflow-supported by workflow-based languages such as BPEL4WS, and (iii) (semi)automatic and interactive testing (e.g. performance) of those combined services and components is of paramount importance. This paper presents the underlying framework as well as an implementation of a workflow-based composition and testing system for services.

The reminder of the paper is structured as follows. In section 2 we discuss composition approaches by presenting fundamental issues of component models and Web services workflows. Section 3 introduces our prototype system, the Component Workbench and its support for workflow-based composition and testing of combined e-services and components, supporting BPEL4WS primitives. Section 4 provides some early suggestions for an increasingly important field of e-services, namely interactive and (semi)automatic testing of component and Web services enactments. Finally, section 5 concludes the paper.

2 Composition Approaches

2.1 Web services

Web services can be seen as a newly emerging distributed computing model for Service Oriented Computing. The standardization process is driven by the growing need to enable business-to-business (B2B) interactions on the Web. Web services are self-contained, self-describing modular applications. Web services introduced a componentized view of web applications and is becoming the emerging platform for distributed computing. The architecture considers a loosely integrated component model, where a Web-Service interface (component) encapsulating any type of business logic is described in standardized interface definition language, the Web services Description Language (WSDL) [1]. Web-Service components interact using an XML messaging protocol and interoperate with other components using communication protocols including the Simple Object Access Protocol (SOAP) [2]. Many software vendors and a plethora of standardization consortia, e.g. ebXML [3], W3C [3, 4], OASIS[2], are providing models, languages, and interfaces for the life cycle of Web services: describing, publishing, unpublishing, discovering, and making them available to users for invocation.

2.2 Component Models and Web services Workflows

Software components are an important tool in building software systems. Reusing existing and pre-tested components within projects leads to shorter development cycles, higher quality, and hence reduced costs.

Binary components rely on the black-box reuse paradigm and have advantages compared with source code components that foster white-box reuse. The primary difference is that components built with black-box in mind do not expose internals of the component to clients. In contrary, clients can access a component only with its external interface(s). Furthermore it is not possible to access internal data structures of component instances by clients. These allows that a component that supports a particular external interface can be replaced by versions with corrected functionality or better performance.

In recent years various component models for black-box compositions, such as Sun's JavaBeans, Sun's Enterprise JavaBeans, Microsoft's COM+ components, or OMG CORBA components, have emerged. Microsoft's .NET framework supports components, too. These models define standards for building components, and for interoperability of components within one component model. Furthermore, these models define how a component's external interface is accessed from programming languages. In addition, most of these component models support the access of components across system boundaries, hence enabling remote access to components. To aid developers in building components component models often provide facilities such as transaction monitors, role-based access control, or instance pooling.

2.3 Workflows

A *Process Model* describes the steps that occur in the real world (e.g., the trucks that deliver goods from point A to point B, the schedules and locations of the drivers) and a *Workflow Model* describes the technology interactions that support, interact with, or implement the real world process model (e.g., system X sends request for purchase order to system Y). Business processes based on Web services can be characterised by some ingredients: they specify (a) the potential execution order of operations from a collection of Web services; (b) the data shared between these Web services; (c) involved partners and their roles in the business process; (d) joint exception handling for collections of Web services, and (e) other issues involving how multiple services and organizations participate internally, or between business partners. Those interactions are modeled by exchanging messages between business partners. One can distinguish between a tightly coupled interaction (conversation), meaning that the responses always require the context of the request (e.g. the phrase book type 19" tells nothing useful on its own. Furthermore it is highly relevant to understand who the business partners one is having an interaction (conversation) with, are, since the conversations differ substantially between a business partner and an external project consultant, to name an example.

Generally speaking, one goal of software engineers is to construct applications by composing already existing components including Web services. Composition of Web services can be analysed from two standpoints: (a) Composition in the part-of sense (granularity), i.e. larger part encapsulates Web services (composite) and exposes itself as a Web service. An analogy are method invocations as part of method definition; and (b) Composition in the sequencing sense, i.e. definition of the invocation order of Web services (often called orchestration, choreography, or coordination). In our paper we will use the following working definition: Composition consists of those activities required to combine and link existing Web services (atomic and composite services) and other components to create new processes.

The current basic layer of Web services builds a foundation for a Service-Oriented Architecture and concentrates on single Web services. It lacks methods for composition of many (atomic or composite) Web services into a reliable and dependable business solution supporting an appropriate level of complexity. In traditional workflow systems the selection of tasks is made from a repository [4–6]. It contains tens to a few hundreds of tasks and the selection is humanly manageable. In Web services workflows, in contrast, potentially thousands of Web services are available. It is impossible for a designer to manually browse through all of the Web services available and select the most suitable ones. Thus, this requires the analysis of Web services QoS with operational metrics characterizing the Quality of Service (QoS)[7] that Web services exhibit when invoked. Web services are autonomous; therefore designers cannot identify operational metrics of Web services at design time. However, when composing (build time) a process, it is required to know the Web services operational metrics. This is still an issue of ongoing research.

3 Composition Environment

This section describes the Vienna Component Framework we have built to access components and Web services and the Component Workbench, our graphical composition environment that allows the design of component compositions in an interactive way.

3.1 Vienna Component Framework

The Vienna Component Framework (VCF) [8] allows interoperability and composability of components from different component models such as Enterprise JavaBeans (EJB) [9], CORBA distributed objects [10], or Microsoft COM+ components [11].

VCF provides a single 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 ease 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 models metadata facility to find out about the *operations*, *properties*, and *event callbacks* a component supports. For controlling the lifecycle of a component's instance and each of those kinds of features VCF defines an interface that contains operations for accessing them. The plugin provides implementations for each of these interfaces, if the component model supports the feature, to provide uniform access for the corresponding component model. Once, a component instance has been created instances of the features are created. For instance, for each operation found with the component's metadata an instance of the class that implements `IMethod` is created and initialized with sufficient information to dynamically call the method.

All features can be accessed with feature queries. For instance, it is possible to get all features that have a particular name or all property features. The use of feature interfaces allows a reflective programming style similar to Java reflection or CORBA's Dynamic Interface Invocation. Since this is tedious for programmers VCF also allows the construction of wrapper classes that make the use of these feature interfaces more convenient. However, having access to feature interfaces and supporting arbitrary complex feature queries ease the construction of graphical composition tools such as the Component Workbench.

Table 1 lists various feature interfaces predefined by VCF. Of particular interest is the `IComposite` feature. This allows the construction of components that contain child components. The `IComposite` feature provides methods to add and remove child component instances to or from a component instance. Hence, it is possible to construct nested components that have virtually unlimited nesting depth. In addition, this feature allows to attach additional information to each child component, such as a readable name for a component instance or layouting information for graphical tools.

Feature interface	Description
`ILifecycle`	Controls instantiation/destruction of a component
`IMethod`	Invokes operations of a component instance
`IProperty`	Queries or modifies the state of a component instance
`IEventset`	Allows the registration of callbacks
`IGUI`	Provides a graphical representation of a component
`IComposite`	Supports nested components.

Table 1. VCF Component Plugin Features

New feature interfaces can be defined by new component plugins. However, use of these interfaces is then restricted to this component plugin.

3.2 VCF Connectors

To compose components together, VCF supports connectors that implement interaction between components and Web services. Similar to component plugins, VCF uses connector plugins that implement features for making connections. The implemented feature types, however, are different for VCF component and VCF connector plugins. Table 2 shows the feature types predefined for connector plugins.

Feature interface	Description
`IControl`	Connects and disconnects a connection
`IRole`	Defines a role, and allows to set bindings for the role
`IConnectorGUI`	Draws the connector, i.e. an arrow
`IConnectorInfo`	Information about dependent components

Table 2. VCF Connector Plugin Features

The `IControl` feature is responsible for enabling a connection between the participants. Each connector has one or more end-points. In VCF these are called *role*. The `IRole` interface provides methods to extract information about the role and links such an end-point to a particular *binding*. A binding is a concrete interaction with another component or a Web service. For instance, a binding for a target role can be the call of a method or the modification of a component instance's state. A binding for a source role can be a notification event. Bindings can use other bindings for delivering arguments. For instance, a method call binding can require parameters that are constructed by calling other methods or are just constant values. The `IConnectorInfo` feature delivers information about the components that are linked with bindings. The `IConnectorGUI` feature provides a graphical visualization for the connector.

3.3 Component Workbench

The Component Workbench (CWB) [12] is a graphical environment for component composition. It uses VCF to unify component access. As VCF, the CWB allows composition of components that adhere any of VCF's supported component models.

Components or Web services that shall be used with the CWB are configured with XML files. These components are shown in toolbars and can be instantiated directly on CWB windows. CWB workpane windows are used to model component compositions. These windows correspond to composite components that implement the VCF `IComposite` feature described above. Components and Web services are directly useable and testable within the CWB. Hence, it is possible to modify the state of a component instance with a property sheet that displays all component properties. It is also possible to invoke methods directly within the CWB.

Connections between components make use of VCF connectors. To connect components the roles and their bindings have to be declared with CWB. This can be either done with a wizard that guides the user through every step in defining the roles, or it can be done directly on the workpane. Connections are visualized on the workpanes. Usually this are just lines with arrows denoting the direction of the connection. It is also possible to visualize indirect connections to connect component instances that provide bindings for a connection. Figure 1 shows two composite components of the CWB with various components and connectors.

4 Composition of e-services and components

4.1 BPEL4WS

The Business Process Execution Language for Web services (BPEL4WS) [13] is an XML-based flow language that defines how business processes interact within or between organizations. It replaces and integrates IBM's proposed Web services Flow Language (WSFL) [14] and Microsoft's XLANG [15] proposition, which is currently implemented in BizTalk Server products. The initial BPEL4WS 1.0 specification was jointly proposed by IBM, Microsoft, BEA in August, 2002 and updated in May 2003 by version 1.1. It allows long-running transactions to execute as Web services with support for consistency and reliability. It supports graphs and algebra and addresses abstract and executable processes. BPEL supports compensation-based business transactions as defined by the WS-Transaction specification. Business Processes specified in BPEL4WS are fully executable and portable between BPEL-conformant environments.

BPEL4WS is a block-structured programming language, allowing recursive blocks, but restricting definitions and declarations to the top level. The language defines *activities* as the basic building elements of a process definition. *Structured activities* prescribe the order in which a collection of activities take place. Ordinary sequential control between activities is provided by sequence, switch, and while. Concurrency and synchronization between activities is provided by the flow constructor. Nondeterministic choice based on external events is provided by the pick constructor. It contains handlers for events including message events (onMessage with portType, operation and partner) and timed events such as duration or deadline. The first event handler of a pick element to receive its event will be executed. Process instance-relevant data (containers) can be referred to in routing logic and expressions. BPEL4WS defines a mechanism for catching and handling faults similar to common programming languages, like Java. One may also define a compensation handler to enable compensatory activities in the event of actions that cannot be explicitly undone. BPEL4WS does not support nested process definition. One of the key aspects of Service Oriented Architectures is the support of dynamic finding and binding of services at runtime (e.g. in a repository such as UDDI). However, in BPEL4WS the notion of dynamic finding and binding is not supported directly. These activities need to be modeled explicitly (as activities, i.e. Web services). The tables 3, 4, and 5

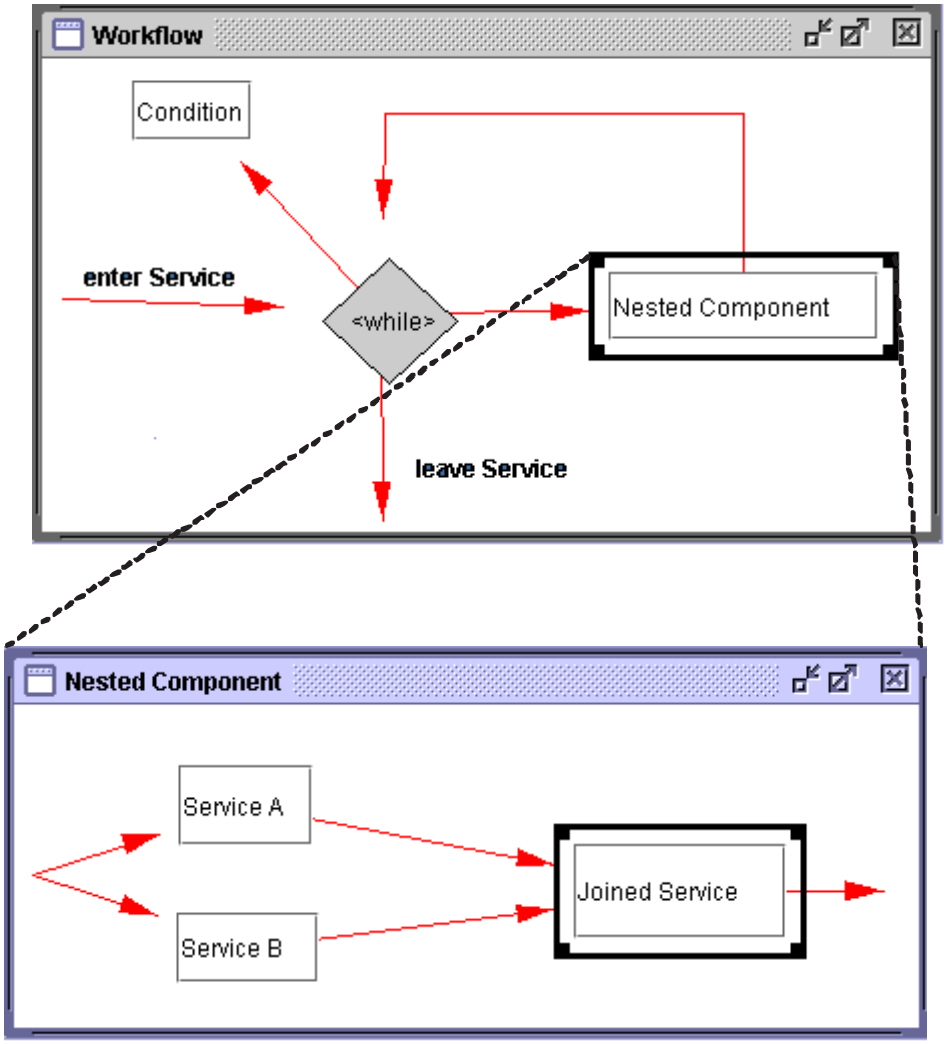

Fig. 1. Composite Components

summarizes BPEL4WS activity primitives. The realization within the CWB is discussed in section 4.2.

BPEL Tags	Characteristics	CWB model
<invoke>	Invokes a Web service (e.g. provided by a partner)	invoke connector
<receive>	Listens to a message at the service interface (i.e. receives message from other Web services)	receive event
<reply>	Generates a response reply for the received activity. Together with the <receive>activity they constitute a request/response pattern.	reply method
<wait>	Enables the process to wait for a defined time.	wait component
<assign>	Variables can be updated (assigned) with new data.	assign component copy connector
<throw> <catch>	The throw activity signals an internal error, which can be catched.	
<terminate>	A process may be terminated explicitly.	terminate event
<empty>	An empty activity, which may be used to work on an error (for example).	empty component

Table 3. BPEL4WS - Primitive Activities

Furthermore, BPEL4WS allows describing relationships (third party declaration) how services interact (what they offer) by introducing *Partner Link Types* (PLNK), with a collection of roles, where each role indicates a list of portTypes. At runtime, the BPEL runtime (execution) engine has to deal with the binding. BPEL4WS also supports the notion of compensation and fault handling. Both concepts are based on the concept of *scopes* (i.e. units of compensation or fault). BPEL4WS creates process instances implicitly, i.e. whenever instances receive a message, an instance is created. This is different to many workflow systems, which identify process instances by their ID. In the case of BPEL4WS any key field, such as an invoice number in an order fulfilment scenario, could be used for this purpose. The BPEL4WS middleware has to deal with. This mechanism is called *message correlation*.

4.2 Modeling BPEL primitives with CWB

A primary goal of this paper was to describe how BPEL can be modeled with VCF and the CWB. To integrate BPEL with CWB we have designed a component model for BPEL workflows and built a corresponding VCF plugin. Since it is possible in the Component Workbench to mix components of arbitrary component models, as long as a VCF plugin is available this allows us to use COM+, or EJB components, CORBA objects or JavaBeans, and Web services that use SOAP for communication.

To design the workflow plugin we have mapped each BPEL4WS activity tag to appropriate VCF constructs, components or connectors that implement

BPEL Tags	Characteristics	CWB realization
<sequence>	Permits sequential invocation of activities.	sequence connector
<switch>	Allows modeling of several branches (such as in programing languages). Alternate route is shown with ¡otherwise¿ clause.	switch connector
<while>	Allows modeling of repetitive activity while a condition is, for example, true.	while connector
<pick>	This construct allows to block and wait for exactly a suitable message to arrive or for a time-out alarm to go off. When one of these triggers occurs, the associated activity is executed and the pick completes.	
<flow>	The flow construct allows the specification of one or more activities to be executed in parallel. Links can be used within parallel activities to define arbitrary control structures.	*implicit*

Table 4. BPEL4WS - Structured Activities

BPEL Tags	Characteristics	CWB model
<scope>	The scope construct allows definition of a nested activity with its own assoated fault and compensation handlers.	
<compensate>	The compensate construct is used to invoke compensation on an inner scope that has already completed its execution normally. This construct can be invoked only from within a fault handler or another compensation handler.	

Table 5. BPEL4WS - Fault Handling

the semantics of the BPEL4WS tag. Although it would have been possible to model each activity as a component and connect these components by connectors (corresponding to the semantics), we have decided to chose different constructs to allow better GUI for modeling business processes with the CWB.

In the CWB components are arranged on workpane windows that map to composite components. We have modified these composite components to allow for BPEL4WS processes. The `receive` activity is the only way to instantiate a business process in BPEL4WS [13]. We have modeled receive activities as events of the composite component. These events can then be connected to the component that represents the first activity to be executed. However, since different receive activities are possible to instantiate a business process, it is necessary to inform the composite component of the set of allowed receive messages. This is done by configuring the composite component in a CWB dialog.

The `reply` activity is the counterpart of a corresponding receive activity and returns a return message to the caller. This activity has been modelled as a

method of the composite component that can be called with the corresponding `receive` event name as arguments.

The composite components itself allow arrangement of arbitrary components. The semantics of putting components in a CWB composite window (section 3.3 is as follows. If a component is put onto the composite component it is executed concurrently. Hence, this implements BPEL4WS's `flow` activity. To build the `sequence` construct we have built a special connector, the `sequence connector`. Activities that are linked with this connector are executed in order of the arrow direction.

The `invoke` activity is modeled as a VCF connector. This allows making a call after a particular service has finished. The source role of this connector is linked to the Web service or the component that is executed before the invocation will happen. The target role is linked to the Web service or the component that shall be invoked.

BPEL variables are modeled as components. The SOAP plugin has already a builtin facility for creating Java record classes that map XML Schema types for SOAP messages. With the JavaBeans plugin the Component Workbench allows the instantiation of these record classes as *variable components*. BPEL variables are scoped. In the CWB this scope is the surrounding composite component that is the parent of the variable component.

The BPEL `assign` activity is modeled as a component, too. The assign activity starts one or more *copy* operations between variables. In the CWB the *assign component* acts as initiator for the copy operations. The copy operation itself is modeled as a connector. Its primary roles are the variable *from* where data is copied from and the variable where data is copied *to*.

The two control flow activities, `while` and `switch` are modeled as connectors. The `while` connector links an evaluation component to a component that shall be executed. Since the latter component can be a composite component again, it is possible to nest arbitrary complex components with `while`. The evaluation component is linked by using the return values of this component.

The `switch` connector connects multiple evaluation components to components that shall be executed. Again it is possible to use composite components to use complex activities.

`wait` is implemented as a component that has a property that stores the duration expression. The termination of a business process is implemented by a `terminate` event that is sent to the outmost composite component.

Currently, we do not support BPEL4WS fault handling. However, it seems to be reasonable to model BPEL4WS `scopes` with composite components and the corresponding `faultHandlers` and `compensationHandlers` similar to the `switch` construct. Futhermore, we currently do not support the `pick` construct.

4.3 Generating BPEL4WS files

The Component Workbench uses XML as storage format for component compositions. We use this storage format to generate BPEL4WS files out of component

compositions. For each of the activites described in section 4.2 the corresponding XML elements are generated. The necessary information to fill XML element and attribute data is either extracted from component properties, or connector role bindings. As described in [16] message types used in Web services can be designed with a wizard. The corresponding WSDL port types can be generated, too. In the same sense, partner links and its types are generated from the CWB components.

In future releases of our tool we want to support re-import of generated BPEL4WS files and also allow the import of BPEL4WS files built with other tools. This will lead to several problems, since we currently do not support all BPEL4WS constructs.

4.4 Testing of Enactments

The Component Workbench supports testing of workflows by modeling test cases with the Component Workbench. For testing workflows we allow to connect monitor components to specific points in the workflow. These points comprise the connectors' roles and ingoing events or outgoing methods of components. The monitor components stop the complete control flow and delegate the control to specific test components. For each test case a set of test components has to be specified and configured with appropriate value sets to test for. The test components can be configured to compare state of components, connectors, and variables with predefined values. If the comparison fails an error has occured in the test case.

Once, a test case has been defined it can either be used interactively, or it is possible to create a Java test class being used with the JUnit testing framework. This test class uses VCF internally and contains all information necessary to create an environment for activating Web services. For each pair of test components and corresponding comparison values a method in the test class is created. The comparisons that are modeled with the CWB are mapped to calls to JUnit *assertion* methods. JUnit supports automated testing of workflows and simplifies regression testing.

4.5 Monitoring of Workflows

In addition to testing of workflows the monitor components can also delegate the flow of control to components that implement monitoring of workflows or performance measurements. We currently do not provide full-fledged components for monitoring the workflow. However, we provide a component that supports tracing of the activation of particular components or connectors.

5 Conclusions

Service oriented Computing is an emerging paradigm for distributed computing. As such it is of paramount importance to provide means for combined service

and component composition. To summarize the contribution of this paper: we present the underlying framework as well as a prototype implementation of a workflow-based composition and testing system for services. Furthermore, we showed how compositions can be workflow-supported as well as visually created by a workflow-based language such as BPEL4WS. Finally we demonstrated how (semi)automatic and interactive testing (e.g. performance) of those combined services and components can be accomplished.

References

1. W3C: WSDL Web Service Description Language. (2001)
2. W3C: SOAP - Simple Object Access Protocol. (2001)
3. OASIS: ebXML - White Paper - Enabling Electronic Business with ebXML. (2000) http://www.ebxml.org/white_papers/whitepaper.htm.
4. van der Aalst, W., ter Hofstede, A., Kiepuszewski, B., Barros, A.: Workflow patterns. In: Distribute and Parallel Databases, Kluwer Academic Publishers (2003) to appear.
5. Benatallah, B., Dumas, M., Fauvet, M.C., Abhi, F.: Towards patterns of web service composition. Technical report, University of New South Wales (2001) UNSW-CSE-TR-0111.
6. Pilioura, T., Tsalgatidou, A.: E-services: Current technology and open issues. In: Proceedings of the 2nd Workshop on Techologies for E-Services (TES'01), Springer (2001)
7. 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)
8. 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)
9. DeMichiel, L.G., Yal cinalp, L.Ü., Krishnan, S.: Enterprise JavaBeans Specification, Version 2.0. Sun Microsystems. (2001)
10. Henning, M., Vinoski, S.: Advanced CORBA Programming with C++. Addison Wesley Longman, Inc. (1999)
11. Kirtland, M.: Designing Component-Based Applications. Microsoft Press (1999)
12. 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 (2003)
13. Oasis: Business Process Execution Language for Web Services Specification. (2003) http://www-106.ibm.com/developerworks/library/ws-bpel/.
14. IBM: Web services Flow Language (WSFL). (2002) http://www-3.ibm.com/software/solutions/webservices/pdf/WSFL.pdf.
15. Microsoft: XLANG Specification. (2002) http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm.
16. Oberleitner, J., Dustdar, S.: Constructing web services out of generic component compositions. Technical report, Vienna University of Technology, Distributed Systems Group (2003) accepted for publication at ICWS-2003 Europe.