Self-Integration of Web Services in BPEL Processes

Steffen Bleul, Diana Comes, Marc Kirchhoff, and Michael Zapf

Kassel University, Distributed Systems, {bleul,comes,kirchhoff,zapf}@vs.uni-kassel.de

Abstract. Interoperability between clients and service may not be seen as a major challenge in SOAs but in reality services under change impose a major hindrance in service management. We present a model and system for service process management where we achieve self-integration by automatic message matching and runtime transformation. We have developed the necessary WSDL schema extension, a semantic discovery algorithm, and a runtime mediation system. Our matching algorithm can detect semantically related message elements and generate appropriate XSL transformations. Finally our system dynamically instantiates mediators to bind services to service processes specified with BPEL4WS.

1 Introduction

Self-integration is an important issue for self-properties inside SOAs. The flexibility of a SOA depends on the ability to add, remove, or update a service without interrupting the course of business. The update of a service and especially the introduction of new services inside the SOA of an enterprise or external services of business partners entail a change in service interfaces.

Service interfaces consist of operations and parameters. It is mandatory that these properties directly correspond to the respective properties of the client's stub; otherwise communication will fail.

By self-integration we refer to the automatic process of integrating Web Services into a SOA by mediation between differing interfaces. This is achieved using a stylesheet transformation (XSLT) [9] which is automatically generated. At first, the semantic cover of two SOAP [7] messages is evaluated. If the messages share semantically equivalent message elements, then the matching algorithm creates an appropriate transformation. Moreover, we introduce a lifecycle for BPEL4WS [2] processes where the necessary services are discovered and integrated using mediators. An integration manager ensures undisturbed functionality of the registered service processes by automatic enforcement of our lifecycle.

The paper is structured as follows. In Section 2 we introduce our model of self-integration in BPEL processes and a service process lifecycle with an integration manager. The self-integration process is described in Section 3 along with the semantic annotation of WSDL [8] and our matching algorithm. Section 4 presents related work, and the paper closes with a conclusion in Section 5.

2 Self-Integration for BPEL Processes

The task of self-integration consists of the automatic binding of applications, which require some service functionality, to a dynamic changing set of services, and furthermore of the automatic deployment of service mediators when necessary. In our approach we not only support process deployment and binding, but also automatic generation of message transformations for incompatible service interfaces. The additional cost is to prepare once a semantic annotation of the message elements inside the WSDL description of a service with OWL [15] individuals (*WSDL + OWL*). The semantic service discovery algorithm enables not only semantic matchmaking [17,5,6] but also produces *XSL transformations* (*XSLT*) for our *message mediation* system. Our system employs runtime transformation of messages with XSL transformations, as well as conversion of values, e.g. data type and currency conversions, with additional software *plug-ins*. The service discovery algorithm in combination with the system allow a high grade of autonomy in relation to service integration without the need of manual administration.

Figure 1 presents some details of our model, focusing on service integration. In our model, services are available when their interfaces description, formulated e.g. in WSDL, are registered inside the *Service Registry*. They are unavailable as soon as their interface descriptions are removed. Descriptions are also removed when the *Monitoring System* detects and throws faults. On the other side, we ensure that BPEL instances are running for all BPEL descriptions inside our *Process Registry*. The central part of our integration system is the *Integration Manager*. The manager is responsible for the failsafe execution of an arbitrary amount of BPEL descriptions by integrating registered services. In principle, the system can handle an arbitrary amount of process registries and service registries, but we limit our model by providing access to only one *BPEL4WS engine* and *Mediation System*.
The manager task processes the following steps:

1. **Service discovery**: The WSDL files of the BPEL description represent service queries and are matched against the service offers. A matching score is calculated and the services are ranked by their matching score.
2. **Service integration**: The discovery algorithm generates XSLT documents for successful matches. Furthermore, the algorithm produces a list of mediator plug-ins which are necessary to convert values. Both the list and the XSLT documents are used for runtime configuration of the *Message Mediation* system and enables ad-hoc service integration.
3. **Deployment process**: The process will be deployed on the *BPEL4WS engine* and their endpoint references are adjusted. They invoke endpoints of *Mediators* instead of direct invocation of the participating Web Services.
4. **Monitoring**: The system removes service descriptions automatically when the monitors discover unreachable services. Afterwards the manager starts anew by discovering services.

Our scenario is a BPEL4WS process of a *book shop* which requires three participating services. The process requests a *storehouse* service for checking available books of a customer's shopping cart, then an accounting service for the creation of receipts.

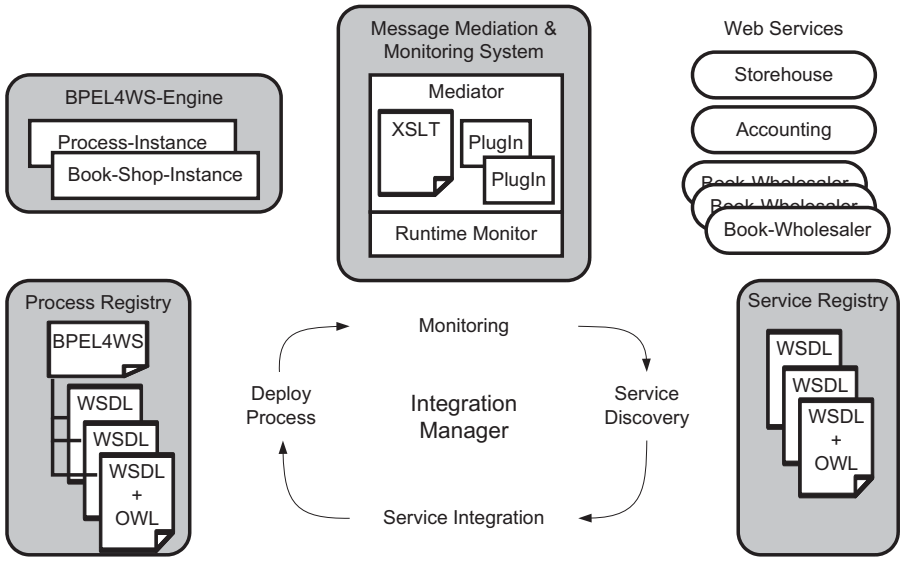

Fig. 1. Self-Integration for BPEL Processes

Additionally we involve a book wholesaler to emulate the need for dynamic changing business partners from a set of book wholesalers. Every service differs in its interfaces from the interface description of the specified business process. Thus, service integration is only possible via message mediation.

3 Automatic Message Transformation for Web Services

The process of automatic message transformation consists of creating transformation instructions and deploying mediators between clients and Web Services. Mediators are responsible for transforming the Web Service request, response, and fault messages. In our approach we achieve a high degree of mediation capabilities just by using matching semantically annotated WSDL descriptions. The WSDL descriptions of a BPEL description are used as Web Service queries and matched against all Web Service descriptions from the service registry.

Mixing both semantic and syntactic enrichments of a WSDL document is significant for our approach which includes an extension of syntactic and semantic information. Furthermore we match aspects like whole XSD element definitions along with data structures, lists of elements, and enumerations with information on how to use the operations interface. A particular strength of the approach is a new degree of matching possibilities which can be achieved with semantics in the case of Web Services:
Interface characteristics: A service interface may have an arbitrary amount of operations. Several operations can have the same functionality. Semantically annotated operations identify related functionality. The operations functionality must be invoked

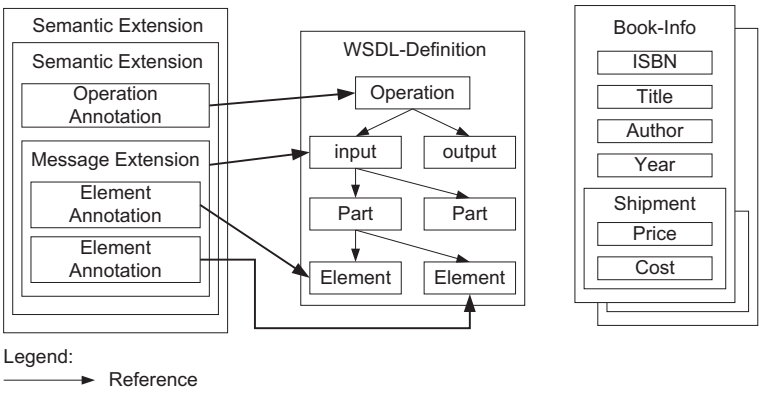

Fig. 2. Structure of an extended WSDL Document.

with a group of parameters. For example, a book search operation can be invoked with both a unique ISBN number or with the tuple title, year, and author. Parameters may be optional or required. Additional syntactic informations give information about service usage and enhance interoperability.

Structural transformation: Seemingly little differences like different parameter names may already prevent interoperability between client and service. The structural transformation deals with the issues of renaming parameters, transforming different structures of complex parameters with sub elements, arrays of parameters and enumerations. Structural differences can be solved by using XSL transformation.

Content transformation: Message values consists of a type, a unit, and a data type. In some cases we need unit conversions, like different currencies or measurements, or data type conversions as required for different date formats. XSLT cannot be used for content transformation of message values. This issue is tackled by a flexible plug-in mechanism for converting components which is automatically integrated and considered by the matching algorithm.

3.1 Semantic Extension of WSDL

As described above we need to define an extension of the WDSL schema in order to achieve the desired transformation features. The extension integrates as a new section in the WSDL with its own document root. Thus, we can add information on WSDL elements without invalidating the WSDL documents. Apart from the automatic message transformation, this information may serve as an additional documentation of Web Service interfaces, with low bounds of initial training of administrators and support tools for interface mediation. An overview is presented in Figure 2.

In the center we give an overview of a structure of a *WSDL definition*. On the left side we present the structure of our *extension section*. A Web Service is described by a set of operations, for each of which we define a *semantic extension*. Furthermore, we define a *message extension* for each input, output, and fault message. A message

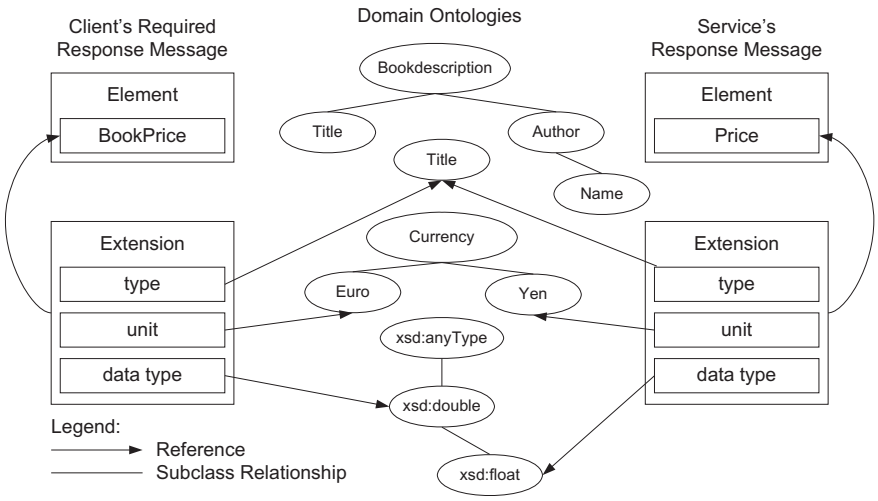

Fig. 3. Annotation of an Element with a Tripplet of Concepts.

is divided into *parts* which are defined by *XSD elements*. The expressiveness of XSD allows to specify message elements with a simple value or a complex element with a subelements. Arrays and enumerations are also part of a schema definition.

Each element may be semantically annotated. In addition to considering input and output parts, we can do reasoning over the whole message structure like the illustrated example on the right side in Figure 2. The example is an informal representation of the response message of a book wholesaler search service. The message has the following structure:

1. On the upper level the response message is an array of elements. The array represents a list of *Book-Infos*.
2. The next lower level is a structure of elements with simple values, e.g. ISBN, title, author, and year. This structure represents the data of a single *Book-Info*.
3. On the lowest level each book has a substructure called *Shipment*. This substructure embeds information about the price of the book and shipping costs.

The structure of the message can be deduced from syntactic information but we need additional semantics to express the meaning of an element.

In our extension, elements are represented by a tuple of concepts representing the type, data type, and a unit. Figure 3 shows an example: On the left side we have an excerpt of the client's response message definition. The client expects an element *BookPrice* whereas the interface of the server returns an element named *Price*. We reference both elements in our extension and annotate them with concepts.

In order to annotate the elements we use three domain ontologies. The upper ontology represents concepts from the *BookDescription* area. The concept *Title* is referenced

by the *type* element of our extension. Both definitions reference this concept, and despite of different identifiers we have a semantic equivalence between both elements.

The next ontology defines concepts of the *Currency* domain. It is used to express the unit of the pricing. While the client's pricing is based on the currency *Euro*, the service returns the price in *Yen*. Here we have a semantic mismatch which requires matchmaking (see Section 3.2).

The last step is the annotation of the data type of the element; here we use the XSD type hierarchies which can be directly transformed into a data type ontology. In this example the client's pricing is expressed by a *double* whereas the server returns the price as a *float*. The semantic cover of the two messages can now be evaluated by our matching algorithm.

3.2 Semantic Service Discovery

Semantic Service Discovery is an algorithm which looks up a service offering the desired functionality. In principle we apply input/output matching of service operations, described in [6]. With respect to this approach, to be more detailed, we are not only looking for a required functionality but we are interested whether the client can invoke the service. This is an issue of interoperability between the client and service. As already mentioned, we have a mediation system which can transform SOAP messages and convert content by means of plug-ins. This must be already considered in the matching process. We cannot fully elaborate on the algorithm in this article, so we give an informal explanation and examples of the key elements of the matching process.

A client invokes an operation of a service by sending a request message in the expected format of the service.

We match the WDSL description of the service as expected by the client against the actual WSDL description of the service and decide if the interfaces are compatible or the discrepancies can be bridged at runtime by our mediation system. The matching process works top-down in the WSDL definition.

First we match operations against each other, comparing input, output, and fault messages. Secondly we check the semantic cover of the messages. An example can be found in Figure 4. On the left side is the expected response message of the client, and on the right side we have the actual response message of the service. We have to match a required message structure against the returned message structure of the following types:

Arrays: An array is a fixed list of subelements. These subelements can again be arrays, complex-types, and enumerations. In order to be compatible the offered message structure must at least deliver one element of the required substructure. In this case they share a semantic cover.

Example: In our example the top-level element is an array of book information. The information itself is represented by a complex type. The required message also has an array as top-level element. Note that even in this case the interoperability may fail due to different identifiers for the array elements.

Complex Types: A complex type is an element that has a fixed defined set of subelements. Complex types can again contain complex types as subelements. The offered message structure must provide all the subelements defined in the complex type in the

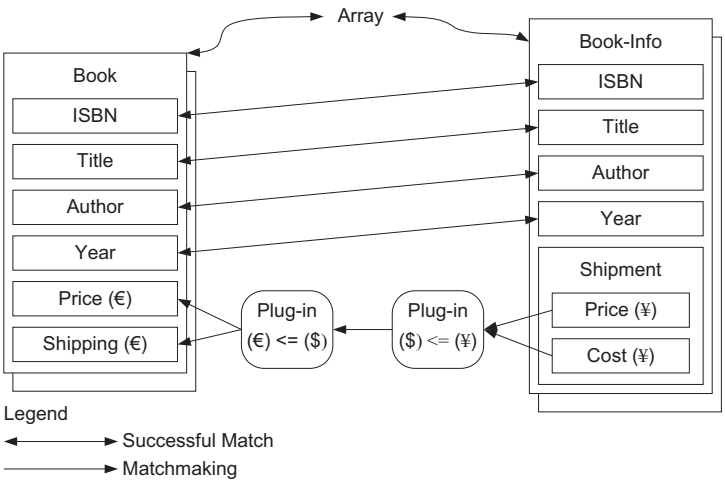

Fig. 4. Semantic Matchmaking Result.

required message structure. This also holds for every subelement which is also a complex type.

Example: The required message structure is a fixed set of simple types. The offered message structure consists of simple types but additionally the shipping data is embedded into another complex type *Shipment*.

Enumerations: An enumeration is a fixed set of simple types. In the concrete message the field for this element may contain a member of this set. The offered message structure must not have more enumerated elements than the requested message structure. All elements of the offered message structure must be semantically related to all elements of the requested message structure.

Example: The genre of the book may be represented by an enumeration. While the requested enumeration may contain the genres romance, fantasy, and research, the offered must not contain more or any other apart from these three genres.

Simple Types: A simple type is an element with a value. The value is described by its semantic type, unit, and data type. The annotated concepts are matched by subclass relationships. The offered simple type must contain a related semantic type with the appropriate unit and a compatible data type.

Example: Here we refer to the example in Figure 3. The requested simple type is on the left side and requires as its value a *Price* in the unit *Euro* expressed by the data type *double*. The offered simple type offers the semantic equivalent type but in the unit *Yen* and is expressed by the data type *float*.

On the lowest level we have the matching of simple types, achieved by matching the semantic concepts for their type, unit, and data type. For specialization relations between A and B, this requires the matching process to consult the domain ontology for the predicate $subsumes(A, B)$ for the concepts A and B: *subsumes* evaluates to true

when A is an equivalent or a subsumption of concept B. In this case, the type annotated with concept B is an equivalent or a specialization of the parameter with concept A. In other words, concept B represents a type that matches the type annotated with concept A or an inheritance of the parameter annotated with concept A. The same applies for the unit and data type of a simple type.

After matching all simple types of the two message structures we get the following result shown in Figure 4. Successful matches are represented by arrows. Both message structures have arrays as their top-level elements. The arrayed book information share a semantic cover because the offered message structure nearly offers all the elements of the required message. It does not offer the necessary elements *Price* and *Cost*, though. Here the recursive algorithm discovers the necessary elements on the next lower complex type *Shipment*. Finally we have discovered the necessary elements but now we discover a mismatch of units for the value of *Price*, *Shipping*, and *Cost*.

At this point matchmaking comes into play. Matchmaking is the process of bridging discrepancies discovered by our matching algorithm in order to have a successful match. Specifically, we apply semantic service discovery itself to dynamically learn about the conversion capabilities of our mediation system. If units and data types do not match, we need content conversion. This conversion is done by plug-ins at runtime.

Here again, we use concepts for specifying the parameters. The algorithm is capable of dynamically discovering plug-ins that can be triggered with the value of the offered message and converted to value of the required message. Moreover, we can discover a sequence of plug-ins as a stepwise conversion from one format into another. In our example in Figure 4, we find two currency conversion plug-ins. The first one converts from yen to dollar and the next one converts from dollar to euro.

The result of the semantic discovery algorithm is generated as a transformation instruction.

3.3 Automatic Transformation Generation

First, our system generates XSLT descriptions; second, it produces a list of sequences of plug-ins. While plug-ins are responsible for content conversion, the XSLT is used to manipulate the XML document inside a SOAP body. Plug-ins are an optional feature; the system may also generate XSL transformations only but will issue a warning about missing plug-ins. This can be used for a semi-automatic support of transformation between Web Services even without our mediation system and improve reusability of our results.

We will now give an overview of the key features of our XSL transformations:
Rename: The simplest key feature is the renaming of identifiers. Messages can have different identifiers for invoked operations, arrays, complex types, enumerations, and simple types. Renaming is also used in case of different namespaces.
Remove: If the offered message structure has more elements or subelements than the required message, this information is removed by elimination of the document nodes.
Copy: Successfully matched elements without differences are simply copied. If the required element is an array but the offered message only offers a structure, this structure is copied as an array. Also, if the required message only requires a complex type but is offered an array, we copy the first element. This is done by matching array bounds.

Cut and Paste: As in our example the shipping information is embedded into a subcomplex type *Shipment*. In this case we cut the element out of the substructures and paste it into the necessary element structure of the required message structure.

4 Related Work

Several projects already deal with semantics for services or even Semantic Web Services. The most prominent ones are OWL-S [13] and WSMO [16]. Semantic extensions for WSDL are available, e.g. WSDL-S [12] and SAWSDL [11]. Our approach is distinct from these concepts in several ways: In the first place our approach is fully automatic from registration to runtime mediation, utilitzing the WSDL annotation. Second, this approach addresses service processes specified in BPEL4WS. Finally, we have implemented a distributed management system featuring self-integration.

OWL-S is an OWL domain ontology for Web Services. It describes the structure, composition, and protocol of a Web Service. The messages are described by a reference on several elements inside the WSDL definition. The OWL-S ontology does not define a domain ontology for parameters and does not extend beyond the definition of WSDL parts. The ontology lacks the definition of fault messages, and it does not define a matching algorithm. Even existing matching algorithms [10,6,3,4] do not match whole data structures, let alone handle automatic generation of XSL transformations.

Service mediation is a key element in the WSMO semantic web services framework. It not only defines message mediation but also mediation between ontologies. If the mapping of parameters is manually specified, the system generates code for runtime mediation. Unlike our system, WSMO lacks a fully automatic mediation. It supports logic components for content transformation but does not apply semantic service discovery using a stepwise transformation of data types and units. Overall, the WSMO framework can mediate without manual administration but requires fixed transformation instructions, e.g. in XSLT.

WSDL-S and SAWSDL both define a schema extension of WSDL for semantic annotation. Whereas WSDL-S is directly related to OWL-S and therefore OWL ontologies, SAWSDL is a more general approach [14]. In WSDL-S additional attributes references concepts in an OWL-S document but in SAWSDL there is the process of uplifting and downlifting. Uplifting is the process of transforming the message's into an ontology or other semantic representation. The reasoner works and transforms on the ontology level which is then transformed back to the required message format.

SAWSDL profits from semantic reasoning on ontologies but does not define a general matching algorithm. Furthermore, uplifting or downlifting for ontologies is done by manually specified transformation descriptions. SAWSDL is complementary to our approach as we only consider subclass relationship in our matching algorithm, but we also offer a general matching algorithm, support transformation generation, and runtime mediation on technologies like WSDL, OWL and XSD definitions.

5 Conclusions

The advantage of a SOA is the flexiblity of integrating new business logic represented by Web Services which are arranged to business processes. However, even if we have a description language like BPEL4WS for Web Services, executing the process is dependent on fixed specified Web Service interfaces. That way we forfeit the flexibility inside a SOA as service process management, which also means time-consuming manual specification adjustments in case of changing interfaces.

In this paper we introduce a model of self-integration for service processes. This self-integration is based on pre-specified WSDL documents which allow to automate the management process from service discovery to runtime message transformation. The WSDL documents of service processes are used as service queries and matched against the service's WSDL descriptions. In case of a successful match, a service is bound to a service process by a Web Service proxy. Even if we have syntactically different message formats between client and service, we are able to mediate as long as there is an XSL transformation between the messages.

Not only may we discover service functionality as in former approaches but also apply a sophisticated matchmaking. The algorithm matches an XSD schema definition of a Web Service and searches for a semantic cover. A semantic cover is found when both messages share enough equivalent elements in their message so that the client can invoke a service and process the response message of the service. The matching result produces XSL transformations and allows a runtime configuration of our mediation system. Moreover, we enhance matchmaking by semantic discovery of additional plug-ins in order to stepwise transform message values on the fly.

The approach as described in this paper has been implemented by us in the course of the ADDO and ADDOaction projects with promising results. The system also includes QoS negotiation and monitoring but this is out of the scope of this paper.

6 Acknowledgments

The work presented here have been funded by the German Research Foundation (DFG) within the project ADDOaction [1].

References

1. Automatic Service Brokering in Service oriented Architectures, Project Homepage. URL: http://www.vs.uni-kassel.de/research/ADDO/.
2. Tony Andrews, Francisco Curbera, Hitesh Dholakia, Yaron Goland, Johannes Klein, Frank Leymann, Kevin Liu, Dieter Roller, Doug Smith, Satish Thatte, Ivana Trickovic, and Sanjiva Weerawarana. *BPEL4WS, Business Process Execution Language for Web Services Version 1.1*. IBM, 2003.
3. Steffen Bleul, Thomas Weise, and Kurt Geihs. Large-Scale Service Composition in Semantic Service Discovery. In *IEEE Joint Conference on E-Commerce Technology(CEC ' 06)and Enterprise Computing, E-Commerce and E-Services(EEE ' 06)*, pages 427–429. IEEE Computer Society, June 2006.

4. Steffen Bleul, Thomas Weise, and Kurt Geihs. Making a Fast Semantic Service Composition System Faster. In *IEEE Joint Conference on E-Commerce Technology(CEC ' 07)and Enterprise Computing, E-Commerce and E-Services(EEE ' 07)*, pages 517–520. IEEE Computer Society, 2007.
5. Steffen Bleul, Michael Zapf, and Kurt Geihs. Automatic Service Process Administration by Semantic Service Discovery. In *7th International Conference on New Technologies of Distributed Systems*, Marrakech, Maroc, June 2007.
6. Steffen Bleul, Michael Zapf, and Kurt Geihs. Flexible Automatic Service Brokering for SOAs. In *Proceedings on 10 th IFIP / IEEE Symposium on Integrated Management (IM 2007)*, Munich, Germany, May 2007.
7. Don Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Frystyk Nielsen, Satish Thatte, and Dave Winer. Simple Object Access Protocol (SOAP) 1.1. W3C Note NOTE-SOAP-20000508, World Wide Web Consortium, May 2000.
8. Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web Services Description Language (WSDL) 1.1. W3c note, World Wide Web Consortium, March 2001.
9. James Clark. XSL Transformations (XSLT). W3c:rec, W3C, November 1999. http://www.w3.org/TR/1999/REC-xslt-19991116.
10. Michael C. Jaeger, Gregor Rojec-Goldmann, Christoph Liebetruth, Gero Mühl, and Kurt Geihs. Ranked Matching for Service Descriptions using OWL-S. In *Kommunikation in verteilten Systemen (KiVS 2005)*, Informatik Aktuell, pages 91–102, Kaiserslautern, Germany, February 2005. Springer Press.
11. Jacek Kopecký, Tomas Vitvar, Carine Bournez, and Joel Farrell. SAWSDL: Semantic Annotations for WSDL and XML Schema. *IEEE Internet Computing*, 11(6):60–67, 2007.
12. Ke Li, Kunal Verma, Ranjit Mulye, Reiman Rabbani, John A. Miller, and Amit P. Sheth. Designing Semantic Web Processes: The WSDL-S Approach. In Jorge Cardoso and Amit P. Sheth, editors, *Semantic Web Services, Processes and Applications*, volume 3 of *Semantic Web And Beyond Computing for Human Experience*, pages 161–193. Springer, 2006.
13. David Martin, Mark Burstein, and Grit Denker et al. *OWL-S, OWL-based Web Service Ontology*, 2004.
    URL: http://www.daml.org/services/owl-s/1.1/.
14. David Martin, Massimo Paolucci, and Matthias Wagner. Bringing Semantic Annotations to Web Services: OWL-S from the SAWSDL Perspective. In Karl Aberer, Key-Sun Choi, Natasha Noy, Dean Allemang, Kyung-Il Lee, Lyndon J B Nixon, Jennifer Golbeck, Peter Mika, Diana Maynard, Guus Schreiber, and Philippe Cudré-Mauroux, editors, *Proceedings of the 6th International Semantic Web Conference and 2nd Asian Semantic Web Conference (ISWC/ASWC2007), Busan, South Korea*, volume 4825 of *LNCS*, pages 337–350, Berlin, Heidelberg, November 2007. Springer Verlag.
15. Deborah L. Mcguinness and Frank van Harmelen. OWL Web Ontology Language Overview. W3c note, World Wide Web Consortium, February 2004.
16. Dumitru Roman, Uwe Keller, Holger Lausen, Jos de Bruijn, Rubén Lara, Michael Stollberg, Axel Polleres, Cristina Feier, Cristoph Bussler, and Dieter Fensel. Wsmo - web service modeling ontology. In *DERI Working Draft 14*, volume 1, pages 77–106, BG Amsterdam, 2005. Digital Enterprise Research Institute (DERI), IOS Press.
17. Thomas Weise, Steffen Bleul, and Kurt Geihs. Web Service Composition Systems for the Web Service Challenge - A Detailed Review. Technical Report 34-2007111919638, November 2007. Permanent Identifier: urn:nbn:de:hebis:34-2007111919638.