Service Offerings for XML Web Services and Their Management Applications

by

Vladimir Tosic, Dipl. Ing., M. Sc.

A thesis submitted to the Faculty of Graduate Studies and Research

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy (Ph.D.)

in Electrical Engineering

Ottawa-Carleton Institute for Electrical and Computer Engineering

Department of Systems and Computer Engineering

Faculty of Engineering

Carleton University,

1125 Colonel By Drive

Ottawa, Ontario, Canada, K1S 5B6

August, 2004

© 2004, Vladimir Tosic

Abstract

I show why the specification, monitoring, and dynamic (run-time) manipulation of classes of service are useful for XML (Extensible Markup Language) Web Services and how to achieve them. Classes of service are a mechanism for differentiation of service and quality of service (QoS) that incurs less overhead than custom-made Service Level Agreements (SLAs), user profiles, and other alternatives. A service offering is a formal description of one class of service of a Web Service. One Web Service can provide multiple service offerings.

For formal specification of service offerings, I developed the Web Service Offerings Language (WSOL), compatible with the standard Web Services Description Language (WSDL). A WSOL service offering can contain descriptions of various categories of constraints, management statements, and reusability constructs. Dynamic relationships between service offerings are specified outside WSOL service offerings, in a special format. Describing a Web Service in WSOL enables monitoring, metering, and management of Web Services and their compositions. The main distinctive characteristics of WSOL, compared to related works, are its expressive capabilities, features with relatively low run-time overhead, and support for management applications.

To achieve dynamic adaptation of a Web Service composition without breaking it, I developed algorithms and protocols for switching (consumer- and provider- initiated), deactivation, reactivation, deletion, and creation of service offerings. I analytically and experimentally compared these manipulation mechanisms with re-composition of Web Services and re-negotiation of SLAs and concluded that the proposed mechanisms are, in principle, simpler, faster, and with lower run-time overhead.

The Web Service Offerings Infrastructure (WSOI) measures and calculates used QoS metrics, evaluates WSOL constraints, and performs accounting of executed operations and evaluated constraints. These monitoring activities can be performed by providers, consumers, or management third parties (SOAP intermediaries or probes). One of the main differences between WSOI and the other Web Service management infrastructures are the modules, data structures, and specialized management operations implementing the mechanisms for manipulation of service offerings. The WSOI prototype extends the open-source Apache Axis SOAP engine. Its additional overhead is relatively low.

The results of this research can be used in future Web Service standards and products.

Acknowledgements

I sincerely thank my Ph.D. supervisor, Professor Bernard Pagurek, as well as the other members of my Ph.D. dissertation committee, for their support, encouragement, guidance, and advice throughout the Ph.D. process. Colleagues and friends with whom I worked and/or discussed scientific or organizational topics were also a valuable source of wisdom. In this respect, I must emphasize Kruti Patel and Wei Ma. In addition, anonymous reviewers of academic papers provided several useful comments, criticisms, and suggestions. On the personal side, I am greatful to my wife for her love, support, and patience. Last, but not the least, my parents, family members and in-laws, teachers and professors, supervisors and mentors, tutors and counsellors, colleagues and friends were indispensable in developing my thirst for knowledge, work ethics, and perseverance.

Table of Contents

ABSTRACT II

ACKNOWLEDGEMENTS...IV

TABLE OF CONTENTS..V

LIST OF FIGURES IX

LIST OF TABLES XI

LIST OF ACRONYMS AND ABBREVIATIONS XIII

GLOSSARY OF IMPORTANT TERMS..XVII

1 INTRODUCTION.. 1

1.1 Web Services 1

1.2 An Overview of Web Service Technologies and Their Importance 3

1.3 Web Service Management (WSM) and Web Service Composition Management (WSCM).. 9

1.4 Motivation for This Research... 12

1.4.1 What Is Needed for Management of Web Services and Their Compositions?... 12

1.4.2 Motivation for Researching Classes of Service for Web Services 17

1.5 Research Goals and Research Questions...................................... 19

1.6 Collaboration with Kruti Patel and Wei Ma................................ 26

1.7 Notational Conventions and Dissertation Organization 27

2 THE USEFULNESS OF SERVICE OFFERINGS FOR WEB SERVICES 29

2.1 Possible Approaches to Comprehensive Service Description and Differentiation 29

2.2 Using Classes of Service for Web Services 35

2.3 The Concept of a Service Offering 38

2.4 Service Offerings versus Alternatives 40

3 WEB SERVICE OFFERINGS LANGUAGE (WSOL) 44

3.1 The Requirements I Placed on WSOL.. 45

3.2 WSOL and WSDL 46

3.3 Concepts in WSOL 47
3.3.1 Service Offerings 49
3.3.2 Constraints and Expressions.. 52
3.3.3 Management Statements 59
3.3.4 Reusability Elements and Attributes... 64
3.3.5 Service Offerings Dynamic Relationships (SODRs) 74

3.4 How WSOL Supports Management Activities............................. 77

3.5 WSOL Tools.. 82

3.6 Comparisons of WSOL with Related Work 86
3.6.1 Related Languages for Custom-made SLAs for Web Services 88
3.6.2 Related Languages with the Concept of Classes of Service for Web Services... 95
3.6.3 Related Languages Intended for Discovery and Selection of Web Services... 98
3.6.4 Other Related Languages...101
3.6.5 Benefits, Advantages, and Original Contributions of WSOL.105

4 DYNAMIC ADAPTATION MECHANISMS USING MANIPULATION OF SERVICE OFFERINGS 109

4.1 Possible Approaches to Dynamic Adaptation of Web Service Compositions 109

4.2 Dynamic Adaptation Mechanisms Using Manipulation of Service Offerings 112
4.2.1 Initial Selection of a Service Offering 114
4.2.2 Consumer-initiated Switching between Service Offerings 116
4.2.3 Provider-initiated Switching between Service Offerings 123
4.2.4 Deactivation of a Service Offering 125
4.2.5 Reactivation of a Service Offering 127
4.2.6 Deletion of a Service Offering 127
4.2.7 Creation of a Service Offering 128
4.2.8 Other Possible Mechanisms for Dynamic Manipulation of Service Offerings 130

4.3 Service Offering Management (SOM) Port Types 131

4.4 Analytical Comparisons of the Proposed Mechanisms and Their Alternatives 137

4.5 Experimental Comparisons of the Proposed Mechanisms and Their Alternatives 146

4.6 Discussion of Benefits and Limitations of the Proposed Mechanisms and Their Potential Integration with the Alternatives 154

5 WEB SERVICE OFFERINGS INFRASTRUCTURE (WSOI) 156

5.1 The Requirements I Placed on WSOI 156

5.2 Overview of WSOI 158

5.3 WSOI Modules Implementing Monitoring of WSOL Service Offerings 161

5.4 Monitoring of WSOL Service Offerings Using Management Third Parties 171

5.5 WSOI Modules Implementing Dynamic Manipulation of Service Offerings 174

5.6 Comparison of WSOI with Related Work 181

6 CONCLUSIONS AND FUTURE WORK 199

6.1 Summary of Contributions and Their Importance....................199

6.2 Possible Wider Implications – Beyond XML Web Service Technologies...208

6.3 Future Work 211

REFERENCES 215

APPENDIX A....... SUB-PROTOCOLS FOR CONSUMER-INITIATED SWITCHING 232

A.1 Switching Beginning (Initiation) – Sub-protocol 'B'..................232

A.2 Initialization of Parties for the New Service Offering – Sub-protocol 'I'..235

A.3 Finalization of Parties for the Old Service Offering – Sub-protocol 'F' 238

A.4 Relaying (Forwarding) of Requests to the New Accounting Party – Sub-protocol 'R'..241

A.5 Notification of the Consumer and the Provider about the Completed Switching – Sub-protocol 'N' 243

A.6 Total Number of Exchanged SOAP Messages in Consumer-initiated Switching 245

A.7 Possible Optimizations...245

List of Figures

Figure 1.1 Parts of an Example WSDL Description of a Web Service 6

Figure 1.2 An Example Web Service Composition 15

Figure 2.1 Multiple Classes of Service for One Web Service 35

Figure 3.1 WSOL as a Complement to Wsdl.. 46

Figure 3.2 Partial UML Class Diagram for WSOL Concepts................................ 48

Figure 3.3 Parts of an Example WSOL Service Offering 50

Figure 3.4 An Example Service Offerings Dynamic Relationship (SODR).......... 77

Figure 3.5 Compilation of WSOL Files 83

Figure 4.1 A UML Sequence Diagram With an Example Scenario of Consumer-initiated Switching between Service Offerings.. 117

Figure 4.2 System-Level UML Statechart Diagram for Consumer-initiated Switching... 121

Figure 4.3 Example Operations in Service Offering Management (SOM) Port Types 135

Figure 4.4 A) Switching of Web Services; B) Switching of Service Offerings .. 148

Figure 5.1 Position of WSOI Inside a Provider Web Service 158

Figure 5.2 The Most Important Modules in The Web Service Offerings Infrastructure (WSOI) 160

Figure 5.3 An Example Configuration of Handlers and Message Processing inside Provider-side WSOI 167

Figure 5.4 An Example Configuration of Management Third Parties as SOAP Intermediaries.. 171

Figure 5.5 Partial UML Class Diagram of the WSOI Management Information Model 175

List of Tables

Table 4.1 Notation for Management Parties in Protocols for Manipulation of Service Offerings 113

Table 4.2 Explanation of Service Offering Management (SOM) Port Types...... 133

Table 4.3 Explanation of Representative Operations in SOM Port Types........... 134

Table 4.4 Notation Used in Analytic Comparisons of the Proposed Dynamic Manipulation Mechanisms and Their Alternatives....... 140

Table 4.5 Formulas for Switching between Service Offerings and Switching Between Web Services........ 141

Table 4.6 Description of an Experiment Comparing Consumer-initiated Switching between Web Services and Switching between WSOL Service Offerings.. 149

Table 4.7 Results of the Experiment Comparing Consumer-initiated Switching between Web Services and Switching between WSOL Service Offerings.. 149

Table 4.8 Description of an Experiment Comparing Provider-initiated Switching between Web Services and Switching between WSOL Service Offerings.. 152

Table 4.9 Results of the Experiment Comparing Provider-initiated Switching between Web Services and Switching between WSOL Service Offerings.. 153

Table 5.1 Description of an Experiment Measuring Additional Overhead of WSOI When Performing Monitoring of WSOL Service Offerings........ 170

Table 5.2 Results of the Experiment Measuring Additional Overhead Of WSOI When Performing Monitoring of WSOL Service Offerings........................ 170

List of Acronyms and Abbreviations

A2A - Application-to-Application

ADL - Architecture Description Language

API - Application Programming Interface

ARM - Application Response Measurement

Axis - Apache eXtensible Interaction System

B2B - Business-to-Business

BMP - Business Management Platform (Agent in WSCM)

BPM - Business Process Management

BPEL4WS - Business Process Execution Language for Web Services

BS - Buy/Sell Stock (example Web Service)

CG - constraint group

CGT - constraint group templates

CIM - Common Information Model

CORBA - Common Object Request Broker Architecture

COTS - Commercial Off-The-Shelf (software component)

CPA - Collaboration-Protocol Agreement

e-business - electronic business

DAMSC- Dynamically Adaptable and Manageable Service Compositions

DCOM - Distributed Component Object Model

DiffServ - Differentiated Services

DMI - Desktop Management Interface

DOM - Document Object Model

DS - Decision Support (example Web Service)

EAI - Enterprise Application Integration

ETTK - Emerging Technologies Toolkit (IBM)

FA - Financial Analysis (example Web Service)

GPRS - General Packet Radio Service

HP - Hewlett-Packard Company

HTTP - Hypertext Transport Protocol

IBM - International Business Machines Corporation

ID - Identifier

IDL - Interface Definition Language

ISO - International Standards Association

IT - Information Technology

JMX - Java Management Extensions

JVM - Java Virtual Machine

m-Service - mobile Web Service

MB - Megabyte

OASIS - Organization for the Advancement of Structured Information Standards

OCL - Object Constraint Language

OGSA - Open Grid Services Architecture

OGSI - Open Grid Services Infrastructure

P2P - Peer-to-Peer

Ph.D. - Doctor of Philosophy (Philosophiae Doctor)

QDL - QoS Description Languages

QML - QoS Modeling Language

QoS - Quality of Service

QRL - Quality Requirements Language

R-RIO - Reflective-Reconfigurable Interconnectable Objects

RPC - Remote Procedure Call

SDI - Service Deployment Information

SLA - Service Level Agreement

SLO - Service Level Objective

SLS - Service Level Specification

SN - Stock Notification (example Web Service)

SO - Service Offering

SOA - Service-Oriented Architecture

SOAP - (formerly) Simple Object Access Protocol

SODR - Service Offerings Dynamic Relationship

SOM - Service Offering Management

TC - Technical Committee

TINA - Telecommunications Information Networking Architecture

UDDI - Universal Description, Discovery, and Integration

URI - Uniform Resource Identifier

UML - Unified Modeling Language

W3C - World Wide Web Consortium

WSB - Web Service Broker

WSCM - Web Service Composition Management

WSDL - Web Services Description Language

WSDM - Web Services Distributed Management (OASIS)

WSFL - Web Services Flow Language

WSLA - Web Service Level Agreement (IBM)

WSM - Web Service Management

WSML - Web Service Management Language (HP)

WSMN - Web Services Management Network (HP)

WSOD - Web Service Offering Descriptor

WSOI - Web Service Offerings Infrastructure

WSOL - Web Service Offerings Language

WSRF - Web Services Resource Framework

WSTK - Web Services Toolkit (IBM)

WS-QoS- Web Services Quality of Service

XML - eXtensible Markup Language

GLOSSARY OF IMPORTANT TERMS

In this glossary, I summarize the meaning of some terms that I use frequently. The number in parentheses refers to the section where the left-side term is defined.

accounting party A party which performs all accounting and billing activities for a service offering. (2.3)

applicability domain A WSOL attribute that defines to which operations, port types, ports, and/or Web Services the given WSOL construct applies. (3.3.4)

chain An Axis module that is a pipelined collection of handlers. (5.3)

class of service A discrete variation of the complete service and quality of service (QoS) provided by one Web Service. (1.4.2)

constraint A condition to be evaluated, represents requirements and guarantees. (3.3.2)

constraint group (CG) A named set of service offering items: constraints, statements, and reusability elements. (3.3.4)

constraint group template (CGT) A parameterized constraint group. (3.3.4)

contract Any formal agreement between two or more involved parties. (1.3)

custom-made contract A contract which contains details specific to the consumer and/or explicit information about the consumer. (2.1)

dynamic relationship between service offerings A relationship that can change during run-time, e.g., shows the appropriate replacement service offering. (3.3.5)

extension attribute A WSOL attribute that enables single inheritance of service offerings, constraint groups, and constraint group templates. (3.3.4)

handler An Axis module that processes input, output, and/or fault SOAP messages (5.3)

inclusion element A WSOL element that enables reusing constraints, statements, and constraint groups that have been already defined elsewhere. (3.3.4)

management information The information necessary for performing management activities, as well as the information about the results of management activities. (1.4.1)

management responsibility statement A WSOL statement that specifies which entity has the responsibility for checking a particular constraint, a constraint group, or the complete service offering. (3.3.3)

management third party An entity independent from the provider and the consumer that performs monitoring and management activities. (1.3)

probe A management third party that periodically sends test requests to the provider Web Service and collects management information from these test requests. (1.3)

re-composition of Web Services The most general approach to dynamic adaptation in which some external entity (e.g., management software or human administrator) composes an new Web Service composition after a change of circumstances. (4.1)

Service Level Agreement (SLA) A special type of contract that specifies in business-oriented terms the expected operational characteristics of the relationship between a service customer and a service provider or between two service providers. (2.1)

SOAP intermediary A management third party that intercepts SOAP messages between the consumer and the provider, performs management activities, and sends its management results as part of the intercepted SOAP message. (1.3)

service offering A formal representation of a single class of service for one Web Service. (2.3)

service offerings dynamic relationship (SODR) A special construct used in WSOL to express a dynamic relationship between service offerings. (3.3.5)

statement Any WSOL construct, other than a constraint, that contains important management information about the represented class of service. (3.3.3)

static relationship between service offerings A relationship that show similarities and differences between service offerings that do not change during run-time. (3.3.5)

switching between service offerings Changing which service offering a consumer uses. (4.2.2)

switching between Web Services A special case of the re-composition of Web Services in which the consumer determines that its provider Web Service is no longer appropriate, searches for its replacement, and chooses the new provider. (4.1)

Web Service A software application identified by a URI, whose interfaces and bindings are capable of being defined, described and discovered by XML artifacts and which supports direct interactions with other software applications using XML based messages via Internet-based protocols [Sch02]. (1.1)

Web Service Composition Management (WSCM) The management of Web Service compositions. (1.3)

Web Service Management (WSM) The management of a particular Web Service or a group of Web Services within the same domain of responsibility. (1.3)

Web Service Offering Descriptor (WSOD) A special XML file format designed for describing the order of WSOI-specific handlers used in a particular context. (5.3)

Web Service Offerings Infrastructure (WSOI) A management infrastructure designed to enable and demonstrate monitoring of WSOL-enabled Web Services and dynamic manipulation of WSOL service offerings. (5)

Web Service Offerings Language (WSOL) A language for the formal specification of classes of service, different types of constraint, and management statements for Web Services (1.5)

1 Introduction

The topic of my Ph.D. dissertation is the specification, monitoring, and dynamic (i.e., run-time) manipulation of classes of service for XML (Extensible Markup Language) Web Services. I answer several research questions related to this topic. I show why the specification, monitoring, and manipulation of classes of service are useful for Web Services and how to achieve them.

In this chapter, I give a general introduction to my Ph.D. dissertation. In the first section, I explain the central Web Service concepts used in this Ph.D. dissertation, while in the following section, I give a brief overview of Web Service technologies and their importance. In Section 1.3, I define the concepts related to management of Web Services and their compositions. Further, I explain the general motivation for research in this area and the focus of this Ph.D. research on classes of service for Web Services (i.e., service offerings) in Section 1.4. This discussion is illustrated with motivational examples in the same section. In the following Section 1.5, I state research goals and research questions. Then, I talk about collaboration with Kruti Patel and Wei Ma, Masters students who worked under my mentorship on topics related to this research. In the final Section 1.7, I explain the notational conventions I used and the organization of this Ph.D. dissertation.

1.1 Web Services

The World Wide Web Consortium (W3C) defines a **Web Service** as “a software application identified by a URI, whose interfaces and binding[s] are capable of being defined, described and discovered by XML artifacts and [which] supports direct interactions with

other software applications using XML based messages via Internet-based protocols" [Sch02]. Here, URI means 'Uniform Resource Identifier' and XML means 'Extensible Markup Language'. While some authors use the alternative term 'XML Web Service' and/or do not capitalize the beginning letters 'W' and 'S', I use the variant 'Web Service' in this Ph.D. dissertation. A Web Service can, for example, provide current stock information, buy/sell stock on request, perform financial analysis, return current exchange rates, report current weather conditions, act as an on-line dictionary, etc.

A Web Service can provide several ports (endpoints). Each port implements a particular port type (interface), which is collection of one or more operations. Operations can be input-output, input-only, output-only, or output-input and contain input, output, and/or fault messages.

Web Services can be used for providing services to human end users. However, the primary goal of Web Service technologies is to address the problem of application-to-application (A2A) integration. They can be used for business-to-business (B2B) integration and/or for Enterprise Application Integration (EAI) within companies.

In a Web Service composition, a Web Service can play the roles of a **consumer** (requester, client) and a **provider** (supplier). Hereafter, I assume that a consumer is a Web Service, not a human end user. For example, when a financial analysis Web Service *FA1* uses operations of a stock notification Web Service *SN1*, then *FA1* is the consumer and *SN1* is the provider. A Web Service can at the same time (i.e., in parallel) be a consumer of many Web Services and a provider of many other Web Services, possibly with different characteristics. The composed Web Services can be distributed over the Internet or another network, run on different platforms, implemented in different programming lan-

guages, and provided by different businesses – Web Service **vendors**. Communication between Web Services can be synchronous or asynchronous; based on Remote Procedure Calls (RPCs) using XML or on the exchange of XML documents.

Web Services are an implementation of a more general **Service-Oriented Architecture (SOA)** [Cera02, Sne02]. In this architecture, there are three main roles: a provider, a consumer, and a **directory** (discovery agency, registry, broker), which stores Web Service descriptions. First, the provider publishes descriptions of its services into the directory. Second, the consumer searches the directory to find information about appropriate providers. Third, the consumer binds to a chosen provider to use its services. This is all performed dynamically, not statically. Hereafter, 'dynamic' means 'run-time', while 'static' means 'design-time or deployment-time'. This model of interaction, so called 'publish-find-bind', enables loose coupling of providers and consumers and, thus, increases agility, flexibility, and adaptability of distributed systems.

1.2 An Overview of Web Service Technologies and Their Importance

Several companies and industrial standardization bodies—most notably W3C, the Organization for the Advancement of Structured Information Standards (OASIS), and WS-Interoperability—develop, regulate, and reconcile Web Service technologies. One frequently referenced approach to the architecture of Web Service technologies is described in [IBM01, Fer01]. However, there is no universal agreement what Web Service technologies are necessary or even useful for practical provisioning of Web Services. For some topics, several competing and partially overlapping technologies were developed.

Nevertheless, it is widely accepted [IBM01, Fer01, Cur01, Cera02, Sne02, etc.] that the two main Web Service technologies are:

1. the **Web Services Description Language (WSDL)** for the specification of Web Service functionality (messages, operations, port types), access methods (messaging protocols used), and location of ports; and
2. the **SOAP (a.k.a. Simple Object Access Protocol) protocol** for XML messaging.

Both WSDL and SOAP are standardized by the W3C. The **Universal Description, Discovery, and Integration (UDDI)** directory for discovery of Web Services, standardized by the OASIS, is often mentioned as the third main Web Service technology, but not everyone agrees about this. I now give a brief overview of Web Service technologies to the extent necessary for understanding of my Ph.D. dissertation and discuss their importance. I assume reader's knowledge of XML.

Web Services Description Language (WSDL)

In this Ph.D. dissertation, I use the WSDL version 1.1 [Chr01, Cera02, Sne02]. Recently, two new versions appeared – WSDL version 1.2 and WSDL version 2.0 [Chi03]. These new versions bring several improvements, but not directly related to the topic of my research. WSDL version 1.1 is still widely used in practice.

The outermost XML element in a WSDL version 1.1 file is <*definitions*>. Its attributes define the name for the WSDL description and namespaces used in the contained elements. This element can contain elements <*import*>, <*types*>, <*message*>, <*portType*>, <*binding*>, and <*service*>. The <*import*> element specifies names of WSDL files the contents of which will be included in the current WSDL file, while the <*types*> element defines data types used in the other elements. The <*message*> element describes a one-

way message. Its attribute is the name of the message, while every contained <*part*> element defines a name and a data type of one message parameter. The <*portType*> element describes a port type as a collection of operations, each described with an <*operation*> element. Both the <*portType*> and the <*operation*> element contain the '*name*' attribute. Inheritance of port types is not supported in WSDL 1.1. An <*operation*> element can contain an <*input*>, <*output*>, and/or <*fault*> element which reference previously described input, output, or fault messages, respectively. The <*binding*> element describes how are operations of a particular port type accessed, i.e., what XML messaging protocol is used. Its attributes are the name of the binding and the name of the described port type. Its contained elements are specific to the used XML messaging protocol. SOAP is the most often used XML messaging protocol. More than one binding can be defined for the same port type. The <*service*> element describes a Web Service as a named collection of ports, each described with a <*port*> element. The <*port*> element has attributes that describe its name and the name of the used binding. Its contained elements are specific to the used XML messaging protocol. For SOAP, an element within the WSDL <*port*> element describes the Internet location of this port.

Figure 1.1 shows the most important parts of the WSDL description of an example Web Service '*buyStockService*', with one port '*buyStockPort*' using the '*buyStockBinding*' binding. This binding uses SOAP to implement the '*buyStockPortType*' port type, which has only one operation '*buySingleStockOperation*'. The input message for this operation is '*buySingleStockRequest*', while the output message is '*buySingleStockResponse*'. This example is a simplification of the example given in [Pat03a].

```
<wsdl:definitions …
 tns:buyStock = ”http://www.buyStockPro.ca/buyStockFile.wsdl” >
  …
  <wsdl:message name = ”buySingleStockRequest” >
    <wsdl:part name = ”symbol” type = ”xsd:string” />
    <wsdl:part name = ”quantity” type = ”xsd:nonNegativeInteger” />
  </wsdl:message>

  <wsdl:message name = ”buySingleStockResponse” >
    <wsdl:part name = ”actualPrice” type = ”xsd:float” />
  </wsdl:message>
  …
  <wsdl:portType name = ”buyStockPortType” >
    <wsdl:operation name = ”buySingleStockOperation” >
     <wsdl:input name = ”input”
      message = ”tns:buySingleStockRequest” />
     <wsdl:output name = ”output”
      message = ”tns:buySingleStockResponse” />
    </wsdl:operation>
    …
  </wsdl:portType>
  …
  <wsdl:binding name = ”buyStockBinding”
   type = ”tns:buyStockPortType” >
    <soap:binding style = ”rpc”
     transport = ”http://schemas.xmlsoap.org/soap/http” />
    <wsdl:operation name = ”buySingleStockOperation” >
      <soap:operation soapAction = ”urn:buyStock-service” />
      <wsdl:input>
        <soap:body use = ”encoded”
         encodingStyle= ”http://schemas.xmlsoap.org/soap/encoding/”
         namespace = ”urn:buyStock-service” />
      </wsdl:input>
      <wsdl:output>
        <soap:body use = ”encoded”
         encodingStyle= ”http://schemas.xmlsoap.org/soap/encoding/”
         namespace = ”urn:buyStock-service” />
      </wsdl:output>
    </wsdl:operation>
    …
  </wsdl:binding>
  …
  <wsdl:service name = ”buyStockService” >
    <wsdl:port name = ”buyStockServicePort”
     binding = ”tns:buyStockBinding” >
      <soap:address location =
       ”http://localhost:8080/soap/servlet/rpcrouter” />
    </wsdl:port>
    …
  </wsdl:service>
  …
</wsdl:definitions>
```

Figure 1.1 Parts of an Example WSDL Description of a Web Service

SOAP

SOAP [Cera02, Sne02] is the protocol for XML messaging standardized by W3C. It used to be an acronym for 'Simple Object Access Protocol', but since this is a misnomer, SOAP is no longer considered an acronym. The SOAP standard defines an XML schema for exchanged messages, data encoding rules, and some conventions for representing Remote Procedure Call (RPC) requests and responses. For this Ph.D. dissertation, only the structure of exchanged messages is of interest, so I summarize it now.

The outermost XML element of a SOAP message is <*Envelope*> and it contains the optional <*Header*> element and the required <*Body*> element. The <*Body*> element contains the actual message to be processed, while the <*Header*> element contains additional information for processing the message. Both elements can contain almost any well-formed XML contents, which gives the flexibility that a SOAP message can carry various information and be used for various purposes. For example, when SOAP is used to carry WSDL messages, the <*Body*> element contains actual values for the WSDL <*part*> elements within one WSDL <*message*> element. When an error occurs, the <*Body*> element contains the optional SOAP <*Fault*> element. The <*Header*> element can contain different elements, called '**headers**' ('header blocks'). For example, special headers can be defined for authentication, session, transaction, routing, or other information. The format of headers is not prescribed by SOAP.

Other Web Service tehnologies

UDDI [Cera02] defines how to build a distributed directory of businesses and Web Services. It defines an XML schema for describing businesses and Web Services, as well as a SOAP-based Application Programming Interface (API) for searching and publishing

UDDI descriptions. A UDDI directory can describe and store any business service, not only Web Services. Consequently, there are some differences in concepts and terminology between UDDI and WSDL.

It is also important to prescribe the flow of information and control between the composed Web Services. Several languages for such description of Web Service compositions were developed, the most popular of which is the **Business Process Execution Language for Web Services (BPEL4WS)** [Pel03].

The importance of Web Service technologies

Easy cooperation and run-time integration of heterogeneous distributed computing and software systems, particularly across corporate boundaries, is a very important topic in the modern business world of increasing importance and pervasiveness of Information Technology (IT), globalization, and mergers/acquisitions. Another important desirable in these environments is faster development, maintenance, and adaptation of complex systems, which can be achieved with componentization and providing software as a distributed service instead of as a shrink-wrapped product. These were some of the main motives for several distributed computing technologies, such as Common Object Request Broker Architecture (CORBA) [Bak97]. Unfortunately, these goals remain elusive, due to various technical and non-technical issues. Web Service technologies are an important step towards achieving these goals. Their main technical advantage is that they leverage already widely used technologies, such as XML for representing structured data and Hypertext Transport Protocol (HTTP) and/or other Internet/Web protocols for message transport. Consequently, companies can relatively easily and cheaply use Web Services with their existing computing and software systems. Further, Web Service technologies

build upon experiences with CORBA distributed objects, Microsoft's Distributed Component Object Model (DCOM), Hewlett-Packard's e-services, Sun's Jini services, software components, and several other software technologies. They are a technical evolution, not a revolution. The main non-technical advantage of Web Service technologies is that they are supported by all major computing companies (IBM, Microsoft, Sun, Hewlett-Packard, Oracle, etc.). Important for success is that these companies push, drive, and market development of Web Service technologies. Further, industrial standardization of these technologies is in progress under auspices of W3C, OASIS, and WS-Interoperability. In addition, these technologies are the basis for several other very promising distributed computing developments, such as Grid Services [Fos02], mobile (m-) Services [Maa04], and Semantic Web enabled Web Services [DAM03]. For all these and other reasons, the Web Service technologies became the most important current distributed computing paradigm and their importance will likely increase in the future.

1.3 Web Service Management (WSM) and Web Service Composition Management (WSCM)

Distributed systems management is the process of monitoring and control of computing systems, networks, and software applications to ensure their regular operation, maximize the quality of service (QoS) they provide to users, discover and fix problems when they happen, accommodate change, keep track of the consumed resources, bill consumers, enforce security, and minimize operational costs. Management activities are often classified into five management functional areas defined by the International Standards Association (ISO): fault, configuration, accounting, performance, and security management [Ram98].

Management activities are important (sometimes business-critical) for many applications of Web Services. For example, some kind of management has to be performed if a used Web Service becomes unavailable. The importance of management of Web Services was discussed in a number of publications, such as [LauT01, Tos01, Far02, vMo02, Kel03].

I classify management activities for Web Services into Web Service Management (WSM) and Web Service Composition Management (WSCM). **Web Service Management** is the management of a particular Web Service or a group of Web Services within the same domain of responsibility. For example, Web Services provided with one application server might be managed as a group. Similarly, Web Services provided by the same business entity might be managed in a unified manner as a group, either completely or only in some respects. Web Service Management consists of the monitoring of the operation of the Web Service and the control of the Web Service to meet the guaranteed service and QoS. The monitoring of a Web Service includes measurement and calculation of relevant QoS metrics, evaluation of various conditions (requirements and guarantees), calculation of prices and monetary penalties, and accounting of executed operations, measured/calculated QoS metrics, evaluated constraints, and monetary sums to be paid.

Web Service monitoring activities can be performed by the provider Web Service, the consumer, and/or one or more mutually trusted **management third parties** [Lud02, Kel03] – entities (e.g., Web Services) independent from the provider and the consumer. Management third parties can be used to perform monitoring and management actions that the provider and the consumer cannot execute. They can also be used when the consumer and the provider do not trust each other, but both trust some independent and well-known Web Service Management entity. For example, when a consumer does not trust

that its provider would accurately measure achieved QoS, these measurements can be outsourced to one or several management third parties trusted by both the provider and the consumer. Some management third parties can be specialized for the evaluation of conditions (or only particular groups or types of conditions), for the metering and calculation of used QoS metrics (or only particular groups of QoS metrics), or for the accounting and billing activities. Other management third parties might be used for a combination of these activities. Management third parties can act as SOAP intermediaries or as probes. **SOAP intermediaries** intercept SOAP messages between the consumer and the provider, perform their management tasks, and send their management results as part of the intercepted SOAP message. On the other hand, **probes** periodically send test requests to the provider Web Service and collect management information from these test requests. In other words, probes act as test consumers.

On the other hand, **Web Service Composition Management** is the management of Web Service compositions. In a general case, the composed Web Services are distributed over the Internet and provided by different business entities. Some of these business entities might not want to relinquish or outsource control over their Web Services. They often have mutually incompatible and even conflicting management goals. In addition, management of the Internet infrastructure is a very challenging task. Consequently, Web Service Composition Management will usually not be able to involve full Web Service Management of the composed Web Services and management of the Internet communication infrastructure. The emphasis in Web Service Composition Management must be on decisions related to which Web Services are composed and how they interact. Interaction between the composed Web Services can be explicitly and formally described in

various contracts. A **contract** is any formal agreement between the provider, the consumer, and, potentially, management third parties. For example, a contract can contain descriptions of provided operations, guarantees of maximum response time, prices, and/or legal responsibilities. Web Service Composition Management consists of the selection of Web Services to be composed, the negotiation and/or selection of contracts between these Web Services, the monitoring of contract fulfillment, the modification of contracts that are no longer appropriate, and possibly the re-selection of Web Services to adapt to changes. The monitoring of contract fulfillment combines the monitoring of the composed Web Services. Further discussion of how Web Service Composition Management differs from Web Service Management is given in [Tos02c, Esf04].

There has been a lot of recent progress, in terms of both research and development of appropriate products, related to Web Service technologies. While some of these works (e.g., those overviewed in Section 5.6) study management-related topics, a number of topics in this area have not yet been studied completely. This Ph.D. dissertation explores and addresses several of these topics.

1.4 Motivation for This Research

1.4.1 What Is Needed for Management of Web Services and Their Compositions?

At the start of my research—when SOAP, WSDL, and UDDI just appeared—I noticed that the basic Web Service technologies do not adequately support Web Service Management and Web Service Composition Management and that they have to be significantly extended for these purposes. I examined what extensions are necessary to support management activities and how to achieve them. My conclusion was that, similarly to the

management of distributed computing systems and communication networks, three things are necessary to achieve management of Web Services and their compositions:

1. Explicit, formal, and precise description of **management information**. This is the information necessary for performing management activities, as well as the information about the results of management activities. Some examples of management information are: definitions of QoS metrics to be measured, expected ranges of values for the measured QoS metrics, conditions (requirements and guarantees) to be evaluated, prices for successful execution of operations, description of what happens if the conditions are not met, and the actual values of the measured QoS metrics and evaluated conditions. One way to describe some types of management information is to use contracts.
2. Definition of **management algorithms and protocols**, which process and exchange the management information and perform management actions. Algorithms are executed within one management party, while protocols address coordination of multiple management parties.
3. A **management infrastructure** that monitors, measures (meters), and controls the values of the described management information and implements the management algorithms and protocols.

Let me explain these three topics. Appropriate specification of management information is necessary for successful management of computing systems, networks, and software applications [Sah02a, Tos98]. While WSDL descriptions are mandatory for using Web Services, they are not enough for Web Service Management and Web Service Composition Management. They do not contain the above mentioned examples of management information, which are needed for many management activities. For example, for

many performance management activities it is necessary to formally and precisely specify what QoS metrics to measure or calculate, which QoS requirements and guarantees to evaluate, when to perform these measurements and evaluations, what parties perform particular management activities, and what happens when conditions are met or not met [Sah02a, Kel03, Tos04]. In business-to-business scenarios in which companies need not trust each other, it is particularly important to use contracts containing management information that can be used as the basis for monitoring, measurement, control, accounting, and billing activities.

While some types of management information prescribe what management actions to perform, it is also necessary to define management algorithms and protocols describing how these management actions are performed. They can be developed for various activities, such as collection and exchange of management information monitored during run-time. One important research area that was not previously studied for Web Services is definition of algorithms and protocols for **dynamic adaptation**. These algorithms and protocols specify what actions to perform after a run-time change occurs. On the Internet, changes can occur suddenly and relatively frequently, particularly in business-to-business scenarios. The goal is to minimize potential negative consequences of the change and/or maximize positive consequences. For example, a dynamic adaptation algorithm or protocol can describe what should a consumer do if its provider Web Service becomes unavailable. Definition of a management algorithm or protocol has to include specification of used operations and the order of these operations. For management protocols, it is also important to state roles of management parties. Additional definitions, such as of used data structures, might be needed.

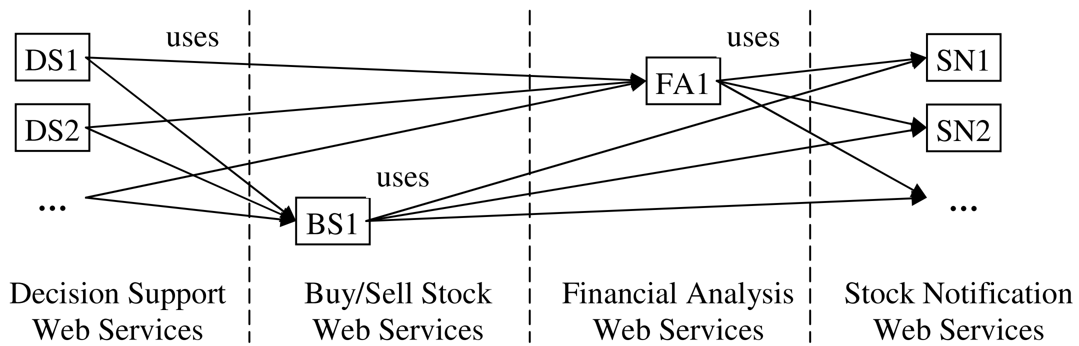


Figure 1.2 An Example Web Service Composition

The development of Web Service technologies is accompanied by the development of appropriate tools and infrastructures, such as application servers that can host (provision) Web Services. However, these infrastructures do not fully support management activities, e.g., they do not measure and collect run-time management information and/or do not perform dynamic adaptation. Without an appropriate management infrastructure, definitions of management information and management algorithms and protocols have only theoretical importance. Implementing management infrastructures as add-ons to well-established infrastructures for hosting of Web Services enables easier addition of manageability to the existing Web Services.

An example Web Service composition and some management scenarios with it

Figure 1.2 shows an example e-business Web Service composition that I use to motivate my work on the specification, monitoring, and manipulation of service offerings. This example will be referenced and extended in the following chapters. In this figure, the financial analysis Web Service *FA1* is a consumer of several stock notification Web Services *SN1*, *SN2*, etc. On the other hand, the financial analysis Web Service *FA1* provides its results, such as recommendations, to different consumers, e.g., decision support

Web Services *DS1*, *DS2*, etc. The financial analysis Web Service can provide these results on request from its consumer, but a consumer can also subscribe for periodic reports from the provider. The financial analysis Web Service can be paid for every invocation separately, using the pay-per-use payment model. Alternatively, it can be paid for invocations during a particular period, using the subscription payment model. There can be more than one financial analysis Web Service, e.g., provided by different vendors and/or performing different types of financial analysis. The decision support Web Services also uses a Web Service for buying and/or selling stock *BS1* or several such Web Services. The latter Web Service uses stock notification services, potentially the same as the ones used by the financial analysis Web Service. To implement the described example, it is enough that these Web Services use infrastructures that support SOAP and WSDL.

I now illustrate the need for the formal specification of management information, management algorithms and protocols, and management infrastructure by adding to this e-business example scenarios that cannot be accommodated with the basic Web Service technologies and the corresponding tools. Since QoS and price can be very important in a business setting, vendors of stock notification Web Services *SN1* and *SN2* decide to formally describe their price and QoS. Assume that they can provide information about the same stocks, but *SN1* provides cheaper service, while *SN2* provides higher availability and lower response time. Further, the decision support Web Service *DS1* requires only simple financial analysis from *FA1* and wants low price, while *DS2* can need either simple or powerful financial analysis and is ready to pay more for additional QoS for any analysis. Consequently, *FA1* formally specifies different QoS guarantees and prices to these two consumers. To achieve these guarantees, *FA1* uses *SN1* for its services to *DS1*,

but uses *SN2* for services to *DS2*. After some time, *SN1* becomes unavailable and *FA1* applies dynamic adaptation algorithms and protocols so that its service to *DS1* is minimally affected. All Web Services use management infrastructures that monitor service and QoS they receive from providers and deliver to consumers. Further, management infrastructures for *FA1* and *DS1* implement dynamic adaptation algorithms and protocols.

1.4.2 Motivation for Researching Classes of Service for Web Services

When a provider Web Service collaborates with a large number of different consumers in parallel, it often has to provide different levels of QoS to accommodate different characteristics and needs of its consumers. Measured QoS metrics, prices, used management third parties, and other management information can also differ between the consumers.

One way to address this differentiation of service and QoS and to describe some management information is to enable Web Services to provide different classes of service to different consumers. By a '**class of service**' I mean a discrete variation of the complete service and QoS provided by a Web Service. Classes of service of one Web Service refer to the same WSDL description, but differ in management information, such as QoS guarantees and prices. In addition, manipulation of classes of service can be used for dynamic adaptation activities in both Web Service Management and Web Service Composition Management. I use the term '**service offering**' to denote a formal representation of a single class of service of one Web Service. It is a synonym for the more cumbersome term 'class of service for a Web Service'. In this subsection, I present a general motivation for researching and using classes of service for Web Services. A detailed discussion of the concepts of a class of service and a service offering will be given in Chapter 2.

While the concept of classes of service is well-know in other areas (e.g., telecommunications), prior to my Ph.D. research there was no work on classes of service for Web Services. As I will discuss in Section 2.2, the prior results from other areas could not have been directly applied to Web Services. Consequently, the focus of my research is enabling Web Services to provide and specify multiple classes of service and to perform management activities, such as monitoring and dynamic adaptation, with them.

Examples of benefits of service offerings and their manipulation

For the financial analysis Web Service *FA1* from Figure 1.2, service offerings could accommodate different classes of consumer, for example decision support Web Services that require a slightly different emphasis or depth of the financial analysis. Also, the service offerings could differ in verbosity of results, the rate of unsolicited notification to consumers, in priority of notification of market disturbances, in the guaranteed response time, etc. Service offerings would differ in price and maybe in the payment model. Thus, they play an important role in balancing the resources used by the financial analysis Web Service *FA1* when it processes requests from a large number of different consumers. Examples of these resources are processing power, threads, consumed memory, and used stock notification Web Services. The resources are limited and their use incurs some costs. For example, the used stock notification Web Services from other businesses (vendors) have to be paid for, probably according to the received level of service and QoS. In fact, the stock notification Web Services can offer multiple service offerings (differing in the rate of notification, verbosity of provided information, etc.) with different prices. From the point of view of the financial analysis Web Service, choosing which stock notification Web Services and their service offerings to use is tightly related with the service

offerings its own consumers request and with the goal of maximizing the monetary gain for its vendor. Similarly, there are benefits of service offerings for the Web Services for buying and selling stock, as discussed and illustrated in [Pat03a, Pat03b].

I now illustrate the benefits of dynamic manipulation of service offerings on the same example. In a turbulent stock market, the adaptability of the relationships between decision support Web Services, financial analysis Web Services, and stock notification Web Services might be a very valuable feature. For example, depending on the analysis of the current situation, the financial analysis Web Service *FA1* could want to dynamically switch between different service offerings of the stock notification Web Service *SN1*. Also, if the decision support Web Service *DS1* consuming *FA1* wants to adjust the service it gets, this adjustment might require dynamic adaptation of the relationship between *FA1* and *SN1*. If for some reason (e.g., mobility) there are some temporary disturbances of the communication between *FA1* and *SN1*, then the financial analysis Web Service *FA1* might have to temporarily deactivate some of its own service offerings dependent on *SN1*. These service offerings can possibly be reactivated after another change of circumstances or completely deleted if changes are permanent. Dynamic evolution (e.g., hot-swapping) of the stock notification Web Service *SN1* can result in the need to dynamically create new service offerings for this Web Service, but also, as a result, for *FA1*.

1.5 Research Goals and Research Questions

The **primary goals** for this Ph.D. research were:

1. To study the applicability of the concept of classes of service for Web Services. To achieve this research goal, I decided to examine alternative approaches, determine poten-

tial benefits and limitations of classes of service, and perform all other research activities explained in this Ph.D. dissertation.

2. To support management (monitoring, metering, condition evaluation, accounting, dynamic adaptation, and other activities) of Web Services and Web Service compositions by enabling explicit, formal, and precise specification of the following management information: classes of service, various types of constraint, and management statements. To achieve this goal, I decided to design a new description language for Web Services, the Web Service Offerings Language (WSOL).

3. To study and enable mechanisms for dynamic manipulation of classes of service that achieve dynamic adaptation of a Web Service composition without breaking an existing relationship between a provider Web service and its consumer. To achieve this goal, I decided to create appropriate management algorithms and protocols and compare them with alternatives using analytical studies and experiments.

4. To enable monitoring of classes of service for Web Services specified in WSOL and to implement the created algorithms and protocols for dynamic manipulation of classes of service. To achieve this goal, I decided to develop an appropriate management infrastructure, the Web Service Offerings Infrastructure (WSOI).

In addition, I made several **research choices** that constrained the domain of my work:

1. I wanted to make adoption of my solutions by the Web Service community easier. Therefore, I decided that my solutions remain compatible with the widely used Web Service technologies: XML, WSDL, and SOAP. I believe that otherwise they would have only theoretical importance and no chance of practical acceptance.

2. I wanted to provide management activities even for relatively simple provider and consumer Web Services. The simplicity of implementation and use, run-time overhead in terms of processing power and used memory, and speed of adaptation can be important characteristics for many Web Services. While management activities are important for any distributed system and are often critical business activities, they are not always used in practice for various reasons. One of the reasons is that the run-time overhead they introduce can be significantly high. Consequently, I was primarily interested in management solutions with relative simplicity, speed, and low overhead. For my research, I did not assume that Web Services are provided by enterprises that already have complex management frameworks and/or application servers supporting management.

3. Since the specification of management information is the basis for management activities, I envisioned WSOL as the basis for my research of adding manageability to Web Services. I was interested primarily in management applications of WSOL and in support for both Web Service Management and Web Service Composition Management activities. I wanted that WSOL could be also used for selection of Web Services and their classes of service, but this was not my primary goal.

4. I envisioned WSOI as a tool that demonstrates that WSOL and the algorithms and protocols for dynamic manipulation of classes of service can be used for monitoring and management of Web Services and Web Service compositions. The WSOL language is not very useful without a management infrastructure that monitors WSOL service offerings. In particular, my goal for the monitoring of WSOL-enabled service offerings was to enable measurement and calculation of used QoS metrics, evaluation of WSOL constraints, and accounting of executed operations and evaluated WSOL constraints. While

important, the control of the behavior of Web Services to meet their guarantees was not a goal of this Ph.D. research. On the other hand, I needed an appropriate infrastructure to demonstrate feasibility of the algorithms and protocols for dynamic manipulation of WSOL service offerings and to experimentally compare them with alternatives. This connection with the research of dynamic adaptation is also reflected in the previously used, but now obsolete, name for WSOI – 'Dynamically Adaptable and Manageable Service Compositions (DAMSC)'. Therefore, I gave special attention to this part of the WSOI infrastructure.

5. I concentrated on business-to-business (inter-enterprise) Web Service compositions because Web Service technologies were originally advertised as appropriate for business-to-business scenarios. Study of features specific for Web Service compositions used for Enterprise Application Integration within companies were not in my research domain.

Secondary goals for WSOL

Apart from the above primary research goals and research choices that influenced more than one sub-project of this Ph.D. research, I also had the following **secondary goals for WSOL**:

1. To enable unified specification of various categories of constraints and management statements, and multiple classes of service for one Web Service in one language, instead of several separate languages. – It is possible to develop and use separate languages for different categories of constraints, such as functional constraints, QoS constraints and SLAs, and access rights. However, there are benefits of describing various categories of constraints, management statements, and classes of service in one language. There is less overhead in supporting one language for various constraints, such as WSOL, than in sup-

porting several separate languages. This is because syntax of different categories of constraints is similar: Boolean expressions containing arithmetic and other expressions. Also, there are significant similarities in the implementation of monitoring and evaluation of different categories of constraints. Due to these similarities, implementing one monitoring and management infrastructure for various categories of constraints and management statements, such as WSOI, incurs less overhead than implementing several infrastructures corresponding to different specification languages. Further, management statements, such as prices/penalties and management responsibility statements, relate to all constraints and not a particular category of constraints, such as QoS constraints. A unified language for various categories of constraints reduces redundancies and potential incompatibilities that can occur when similar information is described in different ways. In addition, dependencies between constraints from different categories can occur and it is easier to express them in a unified language. This topic was also discussed in [Tos03b, Tos04b].

2. To enable reusable specification of service offerings. – When multiple classes of service, SLAs, or other contracts are specified, there is often a lot of similar information that differs in some details. For example, SLAs that a telecommunication service provider has with its customers often have similarities. Analogously, two classes of service for the same Web Service can be the same in many elements, but differ only in response time and price. Expressive mechanisms for reuse of specifications enable easier definition of new classes of service, SLAs, or contracts from existing ones. In addition, they can be very helpful in determining similarities and differences between two classes of service, SLAs, or contracts in the process of selection of Web Services and their QoS.

Secondary goals for WSOI

Further, I identified the following **secondary goal for WSOI**:

1. To support invocation of algorithms and protocols for dynamic manipulation of service offerings by an **external** management software or human administrator. – I was primarily interested that WSOI empowers the provider Web Service to provision, monitor, and manipulate service offerings **autonomously**, i.e., without intervention by external management entities. However, I also wanted to leave open the possibility that, if needed, humans or management software external to the composed Web Services can be involved in the manipulation of WSOL service offerings. This can be useful when decisions about the manipulation have to take into consideration complex circumstances, such as business-level strategic alliances.

The stated primary research goals, research choices, and secondary goals for WSOL and WSOI were refined into requirements for WSOL and WSOI, explained in Section 3.1 and Section 5.1, respectively.

The list of answered research questions

To give an overview of my contributions to the body of knowledge, let me state that in this Ph.D. research I answer the following **research questions**:

1. Why are classes of service (service offerings) useful for Web Services?
2. What are the main alternatives to service offerings?
3. Why are service offerings more appropriate than alternatives in some circumstances?
4. What is needed to enable specification, monitoring, and manipulation of service offerings?

5. What concepts and features should a specification language for service offerings have?

6. What could be other theoretical contributions of such a specification language, in addition to the support for service offerings?

7. How could monitoring of service offerings be achieved?

8. What mechanisms can be used for exchange of run-time values of management information between management parties?

9. Is the overhead of monitoring of service offerings acceptable?

10. What mechanisms can be used for manipulation of service offerings?

11. What are the precise algorithms and protocols for these mechanisms?

12. How can these algorithms and protocols be implemented?

13. What data structures are useful for storing information related to monitoring and manipulation of service offerings?

14. What externally available operations are needed or useful for monitoring and manipulation of service offerings?

15. What are the main alternatives to the mechanisms for manipulation of service offerings?

16. What methodology can be used for analytical comparisons of different dynamic adaptation mechanisms for Web Services?

17. Which factors influence the delay introduced by various dynamic adaptation mechanisms for Web Services and how?

18. Why are the mechanisms for manipulation of service offerings useful and more appropriate than alternatives in some circumstances?

19. What are the limitations of these mechanisms?

20. What is the recommendation about using service offerings for Web Services and the mechanisms for their manipulation?

1.6 Collaboration with Kruti Patel and Wei Ma

Masters students Kruti Patel and Wei Ma worked under my guidance on topics related to WSOL and WSOI, respectively. This Ph.D. dissertation concentrates on my contributions on the architecture of WSOL and WSOI, but I will occasionally mention how their results integrate with my research. The details of their work are in [Pat03a, Pat03b, MaW04].

After I decided what concepts WSOL would or would not support and developed these concepts and other characteristics and features of the language, Kruti Patel coded the XML grammar for WSOL [Pat03a, Pat03b] using my work, examples, and explanations as input. She also implemented a WSOL parser, called 'Premier', to validate the WSOL grammar. Several results of her work helped me improve WSOL concepts.

For WSOI, I decided what modules would or would not be part of WSOI, how they would relate to each other, as well as other characteristics and features of the infrastructure. In addition to creating the high-level design of WSOI, I designed and implemented almost all WSOI-specific data structures and modules used for manipulation of service offerings. Wei Ma designed and implemented several WSOI prototype modules for monitoring of WSOL-enabled Web Services [MaW04]. While for some modules he mainly followed the requirements and input I provided, for some modules he independently determined requirements and completed all stages of the development. For experimental comparisons of the switching between service offerings and switching between Web Ser-

vices, I specified requirements and outlined the design, while Wei Ma implemented and conducted these experiments using the WSOI prototype. Some results of his work helped me improve the WSOI architecture.

1.7 Notational Conventions and Dissertation Organization

One of the notational conventions is that I use bolded text to emphasize definitions of important concepts and the main characteristics of my work. Names of XML elements in the WSOL grammar and values of data tuples are written between < and >, while names of attributes in the WSOL grammar, names of operations and data members within classes and WSOI modules, values of constants, and terminological phrases are written between ‘ and ’. I italicize text to distinguish names of classes and WSOI modules, as well as everything delimited with < and > or ‘ and ’ except terminological phrases.

The remainder of the dissertation is organized as follows. In Chapter 2, I explain the benefits of classes of service for Web Services and compare them with alternative approaches to comprehensive service description and differentiation. In Chapter 3, I give a detailed explanation of WSOL concepts, how this language supports management activities, and how it compares with related languages that were published in parallel with my Ph.D. research. The algorithms and protocols for the dynamic adaptation mechanisms based on the manipulation of service offerings, as well as their analytical and experimental comparisons with alternatives, are summarized in Chapter 4. Chapter 5 details the architecture of WSOI, its modules for monitoring and for manipulation of service offerings, usage scenarios within a provider Web Service and with management third parties, as well as how WSOI differs from recent related works in the area of management of Web

Services and their compositions. In Chapter 6, I summarize conclusions, explore the wider implications of this Ph.D. research, and outline several topics for future research. Appendices contain some information additional to the one presented in the main chapters. Appendix A contains a detailed explanation of sub-protocols for consumer-initiated switching and how the formulas for the number of the exchanged SOAP messages were generated for these sub-protocols.

2 The Usefulness of Service Offerings for Web Services

In Subsection 1.4.2, I presented a general motivation for researching and using classes of service for Web Services. This chapter explains in detail the benefits of classes of service for Web Services. I start the chapter with an overview of several approaches that are used or can be used for comprehensive description and differentiation of Web Services and QoS they provide. In Section 2.2, I discuss using classes of service for Web Services. In the following section, I discuss the concept of a service offering. I compare classes of service for Web Services with alternative approaches to description and differentiation of service and QoS in Section 2.4.

2.1 Possible Approaches to Comprehensive Service Description and Differentiation

The issue of comprehensive service description and differentiation (customization), while new for Web Services, already appeared many times in software engineering, telecommunications, and various businesses. Several approaches to solving this issue can be observed across various application areas. I outline here several approaches that can be applied to Web Services.

Contracts

A **contract** is any formal agreement between two or more involved parties. Customized contracts are widely used in various businesses. For example, when one company sells electricity or Internet access to another, they sign a contract specifying quantitative and qualitative characteristics of the sold service. The contents of contracts can be very

different. For example, it can include technical details such as voltage or network speed, business details such as price, and legal details such as liabilities. To achieve autonomous contract monitoring, enforcement, and management, it is necessary to have formal and precise machine-understandable representation of contracts [Sah02a]. For example, ebXML (Electronic Business Extensible Markup Language) [OAS04, Ira01] is one attempt to enable formal XML-based specification of contracts for electronic business. In ebXML, a business party defines its Collaboration-Protocol Profiles (CPPs) and negotiates with other business parties contractual Collaboration-Protocol Agreements (CPAs) [OAS02]. However, ebXML Collaboration Protocol Agreements do not provide description of QoS and specification of multiple classes of service. Other examples of formal specifications of contracts are given in [Nea03] and [XuL03]. In many cases, contracts are **custom-made**. This means that they contain details specific to the consumer and/or explicit information about the consumer.

Since the definition of a contract is very broad, contracts for Web Services can contain various information. For example, WSDL files are contracts. As discussed in Section 1.4, additional contracts formally describing QoS, security, payments, and other management-related information are needed for the management of Web Services and Web Service compositions. While ebXML Collaboration Protocol Agreements can be used for Web Services, they do not address a number of issues, the most important of which are comprehensive description and differentiation of QoS.

SLAs

Service Level Agreements are a specialized type of contract that has been used in telecommunications and computing service provisioning (e.g., [Lew99, Hau99]). A **Service**

Level Agreement (SLA) specifies in business-oriented terms the expected operational characteristics of the relationship between a service customer and a service provider or between two service providers, so that they can be monitored and managed [Lew99, Kel03, Sah02a]. Different service providers adopted different templates for SLA specification. In many SLA templates, the details of the operational characteristics are represented within an SLA as service-level objectives (SLOs). For example, Service Level Objectives can define acceptable (guaranteed) and unacceptable QoS levels, as well as used QoS metrics. SLAs are often custom-made and negotiated between service consumers and service providers. Due to many different QoS metrics that can participate in SLAs and the difficulty of comparing their combinations, definition and negotiation of custom-made SLAs can be very complex. While there are relatively recent attempts to automate this process, human involvement is still necessary in most non-trivial cases. As will be discussed in Section 3.6, several projects applied the concept of a custom-made SLA to Web Services.

Policies

Management policies are another approach to comprehensive service description and differentiation. **Policies** are declarative descriptions of what must be accomplished by management [Slo94, Slo95]. A management policy is specified as a set of rules that define desired states and behavior of the managed system and its resources. Policy rules often have the following format [Lew96, Raj99]:

IF <set of conditions to be met (states of managed resources, events in the managed system, time, etc.)>

THEN <set or sequence of actions to be taken when the conditions are met>.

Polices can be specified at different levels of abstraction [Bou97, Lup97, Raj99]. High-level policies are abstract, business-oriented, and expressed in business terms and measures. Thus, they are very close, if not identical, to Service Level Objectives. On the other hand, lower-level policies are more refined, geared to the management of specific managed resources, and expressed in technical terms and measures. Policy implementation is achieved by transforming higher-level policies into consistent lower-level policies, until the bottom level is reached when appropriate management scripts and programs are started. Two frequently mentioned problems with policies are policy refinement and the possibility of conflicts between policies [Lup99]. For example, conflicts can occur if policies are negotiated independently.

While [Hon03] lays ground for the specification of some policies for Web Services, it does not enable comprehensive description of QoS for management purposes. This recent work will also be discussed in Section 3.6.

Consumer (user) profiles

Personalization is customization of service and QoS of software systems primarily to human users, not to other software modules. Among the personalization approaches, the use of user profiles (e.g., [Cin00]) seems to be most applicable to Web Services. One of the papers that discuss the use of profiles for Web Services is [Tho01]. While the term 'user (consumer) profile' is used in different ways, hereafter by a '**consumer profile**' I mean a description of characteristics of a consumer or a group of consumers. In this sense, a consumer can be a human user or a software component, e.g., a Web Service. A consumer profile contains information about the consumer, including its preferences (e.g., QoS preferences) and rights (e.g., in the form of authorization policies). It can also

contain personalized information about the provided service, e.g., prices corresponding to consumer's QoS preferences, as well as various other information.

Note that a consumer profile is a characteristic of a consumer or a group of consumers, not a characteristic of a provider Web Service. In particular, a consumer profile is not a contract or SLA with a particular provider Web Service because it need not precisely describe the provider's obligations and guarantees to the consumer. (This difference between SLAs and profiles is not sharp. It is possible to define profiles in such a way that they contain the same information as custom-made SLAs.) The advantage of consumer profiles is that some might be useful for personalization of service and QoS from different provider Web Services. On the other hand, the disadvantage is that for managing provider Web Services, it is harder to use consumer profiles than SLAs. Similarly to custom-made SLAs, when a provider Web Service has a large number of different consumers, a large number of different profiles have to be stored and satisfied.

Parameterization

Another possible approach to customization of service and QoS is to use parameterization. **Parameters** can be special input and/or output parts of operations or special data members describing service and QoS. For example, description of desired and achieved response time for the operation '*getStockPrice(IN String stockName, OUT NonNegative-Float stockPrice)*' can be achieved by adding the parameters '*IN NonNegativeFloat max-ResponseTime*' and '*OUT NonNegativeFloat actualResponseTime*' into the operation description. A consumer customizes QoS by providing appropriate values for '*maxResponseTime*'. Parameterization has several disadvantages. The main one is the need to change existing functional interfaces to describe additional management information with a new

parameter. Additional disadvantages are complexity, run-time overhead for writing and reading parameters, difficulty in definition of parameters describing and differentiating functionality, difficulty in specification of valid combinations of parameters, and lack of relationships between valid combinations of parameters.

Multiple Web Services, ports, and/or operations

One could also suggest using multiple Web Services, ports, and/or operations for service and QoS differentiation of Web Services. For example, a provider Web Service can have two ports implementing the same operations – one with 'low QoS and low price' and another one with 'high QoS and high price'. While this approach provides service and QoS differentiation, it does not provide comprehensive description of information needed for management of Web Services. In the previous example, there is no explicit, formal, and precise WSDL description of what is 'low' or 'high' QoS or price. Such descriptions require additional specification mechanisms. Due to the lack of such descriptions, monitoring and management of Web Services and Web Service compositions is significantly harder. Another disadvantage is that using multiple Web Services, ports, and/or operations to provide the same functionality with different QoS incurs considerable overhead due to redundancy.

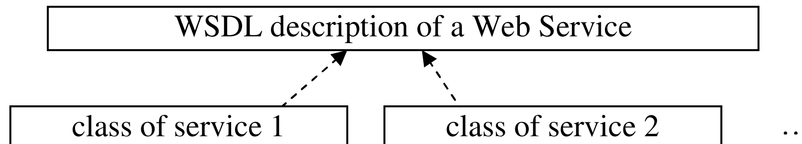


Figure 2.1 Multiple Classes of Service for One Web Service

2.2 Using Classes of Service for Web Services

In addition to the discussed approaches to comprehensive service description and differentiation, another approach can be used – classes of service. As mentioned in Subsection 1.4.2, by a '**class of service**' I mean a discrete variation of the complete service and quality of service (QoS) provided by one Web Service. By 'discrete', I mean that a provider Web Service offers a limited number of classes of service and a consumer chooses between them. I discuss classes of service at the level of Web Services, not at the level of particular constraints or guarantees (e.g., response time) that are part of the overall service and QoS.

In my work, a provider Web Service can offer multiple classes of service to its consumers. This is shown in Figure 2.1. On the other hand, multiple consumers can use the same class of service. This is because classes of service are not custom-made for a particular consumer. Classes of service of one Web Service refer to the same WSDL description, but differ in constraints (capturing requirements and guarantees) and management statements. For example, classes of service can differ in usage privileges, service priorities, response times guaranteed to consumers, and/or verbosity of response information. The concept of classes of service can also accommodate different capabilities, rights, and needs of consumers, including the power and type of devices on which they execute. Further, different classes of service may imply different utilization of the underlying hard-

ware and software resources and, possibly, use of different independent (third) parties for performing management activities. Consequently, different classes of service can have different prices. Additionally, they can be used with different payment models, such as pay-per-use and subscription-based.

To summarize, a provider Web Service with multiple classes of service can be used in different circumstances and by a wider range of consumers. Therefore, providing multiple classes of service enables the broadening of the market segment of a Web Service. It also enables the provider Web Service to better balance limited underlying resources and the price/performance ratio.

Since classes of service describe important management information that is not described with WSDL, their specification, monitoring, manipulation, and use are beneficial for the management of Web Services and Web Service compositions. I will discuss and illustrate management applications of classes of service in Chapter 4 and Chapter 5.

In telecommunications, classes of service are frequently used when it is not possible or economically viable to customize service and QoS to every consumer, e.g., due to a large number of consumers. For example, when a mobile telephone company has a large number of subscribers and cannot customize QoS to all of them, it offers several predefined 'packages' representing classes of service. The concept of classes of service is also used in many other business areas outside information technology. For example, many tourist agencies offer 'package deals' with some variations in quality of service. With such packages, tourists can choose between several hotels with different ratings and prices. These examples show that there is a widely used pattern to achieve service and QoS differentiation with relatively low overhead by providing multiple classes of service.

At the beginning of my research, classes of service were widely used for telecommunications service provisioning, but not for XML Web Services. For example, General Packet Radio Service (GPRS) offers several distinct classes of service. The broader area of comprehensive description and differentiation of service and QoS was extensively researched in telecommunications technologies such as TINA (Telecommunications Information Networking Architecture) [Kri97] and DiffServ (Differentiated Services) [Aim00]. However, these works are specific to the telecommunications domain and are not directly applicable to Web Service technologies [Tos03b]. While telecommunication classes of service sometimes describe characteristics of several QoS metrics, they are often defined for one particular QoS metric (e.g., delay, jitter, or packet loss rate). The systems for monitoring and management of such classes of service are restricted to the QoS at the communication level. The majority of used dynamic adaptation mechanisms, such as rerouting, are not directly applicable to Web Services. On the other hand, classes of service for Web Services can contain much greater variety of management information. Two examples of management information that are not specified on the telecommunication level, but can be important for Web Services, are a percentage of successful transactions and a functional pre-condition. Further, Web Services must be described and discovered using XML, in a way compatible with WSDL. Solutions for Web Services must also take into consideration the heterogeneity of implementations and execution platforms, the application-to-application and business-to-business nature of Web Service compositions, and the goal of the dynamic, autonomous, and Internet-wide Web Service discovery, composition, cooperation, and adaptation. The telecommunications solutions for differentiation of service and QoS address a different execution context.

2.3 The Concept of a Service Offering

I use the term '**service offering**' to refer to a formal representation of a single class of service of one Web Service. Consequently, a service offering contains formal representations of various constraints and management statements that determine the corresponding class of service. Motivational examples for the specification, monitoring, and manipulation of classes of service for Web Services were presented in Subsection 1.4.. The main concept in my Web Service Offerings Language (WSOL), presented in Chapter 3, is the service offering.

Since a service offering formally describes constraints and management statements related to consumers' use of a provider Web Service, it is a type of technical contract. More precisely, many of the constraints and management statements in service offerings are related to QoS and price, so a service offering can be viewed as a special type of predefined (not custom-made) SLA between a provider Web Service, an anonymous consumer, and, possibly, management third parties. However, service offerings can also contain constraints, such as functional constraints, that are not specified in SLAs (nor in classes of service used in telecommunications). In Section 3.3, I will describe in detail service offerings in my WSOL. Then, I will compare WSOL service offerings with custom-made SLAs in major related works in Section 3.6. Note that service offerings in my work are not legal contracts because they do not contain contractual legal aspects. In the future, specification of legal information might be added into service offerings.

Contrary to custom-made SLAs, a consumer and a provider, in principle, do not negotiate the contents of a used service offering. They only negotiate what service offering is

used and this negotiation is very simple – the provider suggests several existing service offerings and the consumer chooses one of them. Input from consumers can, but need not, be used for the creation of new service offerings. Even when a service offering is created with input from a particular consumer, multiple consumers can use it. In many cases, the vendor of the provider Web Service defines its service offerings statically, before the provider starts serving consumers. As I will explain in Subsection 4.2.6, there is also a possibility of dynamic creation of new service offerings.

When a consumer invokes a provider's operation, it is necessary to determine which service offering is used for this invocation. Further, for several monitoring activities—such as measurement or calculation of periodic QoS metrics, evaluation of periodic QoS constraints, accounting of subscription periods, and some others—it is necessary to perform grouping of management information from several consumer's invocations of the provider. Several mechanisms can be used for association between operation invocations and service offerings and/or for grouping of management information. For simplicity, I chose that in my work consumers open sessions with providers. (Note that the contributions of this Ph.D. dissertation can be adapted to alternative mechanisms.) A provider Web Service can have in parallel many open sessions with different consumers. Some consumers might be allowed to open multiple parallel sessions with the same provider Web Service. A provider Web Service can suggest different service offerings to different classes of consumer and even several service offerings to the same consumer. Inside one session, only one service offering is used at a time, so every consumer has to choose among the suggested service offerings. In Subsection 4.2.1, I will describe how a consumer chooses between service offerings of the same provider Web Service. In addition,

the consumer or the provider can initiate a change of service offerings, called switching, and other dynamic adaptation mechanisms, discussed in Section 4.2, that are executed without closing a current session.

In Section 1.3, I defined the concept of a management third party. In my work, the provider, the consumer, and/or one or more management third parties can monitor a service offering. Further, I identified that a special management party is important for my work on service offerings – the **accounting party**. This party performs all accounting and billing activities for a service offering. It is responsible for keeping track of the use of the provider Web Service and management third parties, as well as what constraints were satisfied and what were not. It also calculates prices and penalties to be paid. In addition, the accounting party is the default management entity. In other words, it is responsible for evaluation of all constraints for which management responsibility is not specified explicitly (e.g., through WSOL management responsibility statements discussed in Subsection 3.3.3). A management third party acting as SOAP intermediary, the provider, or, in rare cases, the consumer can play the accounting party role. For simplicity and to prevent potential conflicts, I limited that every service offering has exactly one accounting party.

2.4 Service Offerings versus Alternatives

As discussed in Section 2.1, there are several possible alternative approaches to comprehensive description and differentiation of service and QoS of a provider Web Service. The approaches such as various contracts, custom-made SLAs, policies, consumer profiles, parameterization, and multiple Web Services/ports/operations can be viewed as alternatives to using service offerings. Compared with some of these alternatives, classes of

service have limitations. However, classes of service also have important advantages. I will first discuss the limitations and then the advantages of classes of service. My discussion will particularly emphasize the limitations and advantages of classes of service compared to custom-made SLAs. I consider custom-made SLAs the main alternative to service offerings, because the majority of recent related work on comprehensive description and differentiation for Web Services concentrated on specifying custom-made SLAs for Web Services.

I am aware that the customization of service and QoS through classes of service has limitations. Classes of service enable discrete, **not continuous differentiation** of service and QoS. In addition, classes of service are, in principle, **predefined** and not custom-made. Using classes of service is not as powerful as using custom-made SLAs, consumer profiles, or separate Web Services. As will be discussed in Section 4.5, the dynamic adaptation mechanisms based on the manipulation of classes of service also have limitations. Consequently, classes of service are not a complete replacement for all alternatives in all circumstances. On the other hand, although the overhead of classes of service is generally low, it can be too high for some circumstances.

As mentioned above, the practice of service differentiation in telecommunication service provisioning and a number of other business areas showed that classes of service are relatively **simple to implement and use** and have relatively **low run-time overhead** compared to the more powerful alternatives. In particular, they are simpler and more lightweight (i.e., with less overhead) than custom-made SLAs. For example, if a provider Web Service has 1000 consumers and wants to use custom-made SLAs, it has to provide 1000 such SLAs. On the other hand, it can provide 5 (or similar small number of) classes

of service and let consumers choose between them. The conclusion about relative simplicity and low overhead of classes of service is also supported by the facts that some telecommunications technologies support classes of service and that operating systems offer a limited number of thread priorities. In addition, management of classes of service is generally simpler, faster, and incurs less run-time overhead than alternatives. I will discuss the latter topic in detail in Chapter 4. Another group of advantages of my service offerings differentiates them both from the alternatives and from classes of service used in other areas. First, service offerings in my work can contain formal description of different categories of constraints and management statements, not only QoS and prices. In addition, I added to my work the specification of formal representation of relationships between service offerings. As will be discussed in Subsection 3.3.5, Section 3.4, and Section 4.2, I use these relationships for management of Web Service compositions. Such support does not exist for alternatives to service offerings nor for classes of service in other areas.

I stated in Section 1.5 that one of the secondary goals for my research was providing management activities even for relatively simple provider and consumer Web Services. I identified that for relatively simple Web Services simplicity of implementation and use, run-time overhead in terms of processing power and memory used, and speed can be important characteristics. Because of their low overhead and minimization of the time spent in negotiation with consumers, classes of service seem especially useful when a provider Web Service executes in resource-constrained environments where the overhead of other service description and differentiation approaches can be unacceptably high. Possible examples are mobile and embedded computing systems. To conclude, I see having a limited

number of classes of service as a useful compromise between service and QoS description and differentiation that is too complex for particular Web Services and having no QoS description and differentiation at all.

Consequently, in some cases classes of service can be a simple and lightweight replacement for the alternatives, including custom-made SLAs. Further, even in situations when some more powerful alternative approach to service and QoS description and differentiation is more appropriate for particular circumstances, classes of service can be a useful addition and complement. For example, if there are several provider Web Services implementing the same port types with different QoS, they can still offer several classes of service each. Similarly, if a provider Web Service offers custom-made SLAs to its consumers, it can offer several predefined, non-negotiable SLAs. If such SLAs also describe constraints, management statements, and relationships that are traditionally not present in SLAs, they become similar to service offerings. While such complementary use of several approaches combines benefits, it also increases complexity and overhead, so it should be used carefully.

3 Web Service Offerings Language (WSOL)

As I mentioned in Subsection 1.4.1, to be able to perform Web Service Management and Web Service Composition Management activities, it is necessary to explicitly, formally, and precisely describe management information. WSDL does not enable comprehensive description of important management information for Web Services. In the previous chapter, I explained the benefits of using classes of service (service offerings) for achieving comprehensive description and differentiation of service and QoS of Web Services.

To enable explicit and formal specification of classes of service, various types of constraint, and management statements for Web Services, I developed the **Web Service Offerings Language (WSOL)** [Tos04b, Tos03b, Pat03b, Pat03a, etc.]. WSOL is my solution for the specification of classes of service for Web Services. In addition, the concepts built into WSOL are the basis for my work on the monitoring and manipulation of classes of service for XML Web Services, presented in subsequent chapters.

In this chapter, I present WSOL. I first discuss major functional and non-functional requirements placed on WSOL in Section 3.1 and then give an overview of how WSOL relates to WSDL in Section 3.2. In Section 3.3, I explain the main concepts in WSOL. A discussion how these concepts support various Web Service Management and Web Service Composition Management activities is presented in Section 3.4. Tools for WSOL are outlined in Section 3.5. I compare WSOL with recent related works in Section 3.6.

3.1 The Requirements I Placed on WSOL

Hereafter, by a 'requirement' I mean a characteristic that WSOL MUST have to satisfy the goals for this Ph.D. research. Consequently, most of the requirements I placed on WSOL are direct consequences of the general goals for this Ph.D. research, the choices about the domain of my work, and the secondary goals for WSOL stated in Section 1.5. For example, one consequence of the second primary goal is that WSOL has to contain formal specification of service offerings, various categories of constrains, and management statements. Several other requirements can also be easily concluded from the stated research goals and choices.

In particular, several requirements that I placed on WSOL are consequences of my research choice to make WSOL adoption by the Web Service community easier. These are:

a. WSOL must be an XML-based language. This is a direct consequence of the definition of Web Services as entities described in XML.

b. WSOL must be compatible with and complementary to WSDL. By 'compatible', I mean that they can be used together without special translation. By 'complementary', I mean that WSOL had to reuse (not redefine) information already available in WSDL files and to provide only additional specification of various constraints, management statements, and classes of service for Web Services. As already discussed, WSDL is the standard for the specification of functionality, access methods, and location of Web Services. If Web Services that already support WSDL needed to discard or change existing WSDL descriptions to support WSOL, my language would not be well accepted by the Web Service community.

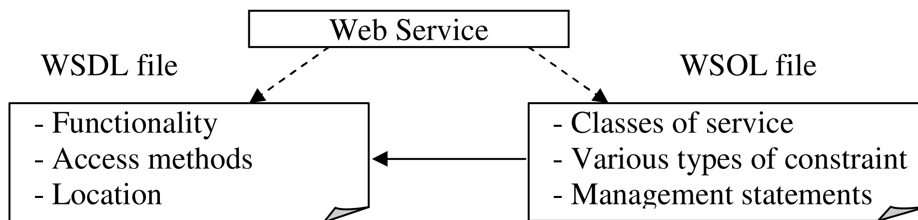


Figure 3.1 WSOL as a Complement to WSDL

c. WSOL must be an optional language for Web Services. By 'optional', I mean that a Web Service need not require from collaborating Web Services to understand WSOL. This is because some Web Services will not need information specified in WSOL files. If a participant does not understand WSOL, only WSDL descriptions are used.

3.2 WSOL and WSDL

The syntax of WSOL is defined using XML Schema. Since I wanted to extend existing Web Service technologies, WSOL is compatible with and complementary to WSDL, but separate from it. This relationship is shown in Figure 3.1. A WSDL file specifies functionality, access methods, and location of Web Services. A WSOL file references one or more WSDL files and specifies additional management information not present in WSDL files. In particular, it adds specification of classes of service, various types of constraint, monitored QoS metrics, and management statements for Web Services.

There was a possibility of implementing WSOL as a WSDL extension. However, it seemed more appropriate to make WSOL a separate language so that, if needed, WSOL service offerings can be deactivated, reactivated, created, or deleted dynamically without any modification of the underlying WSDL files. Such updates may be needed during the management of Web Services and their compositions. For example, while some con-

straints and management statements (particularly functional constraints) rarely change during run-time, other constraints and management statements (e.g., QoS constraints and prices/penalties) can be changed during run-time to better fit the execution circumstances.

There are three main versions of WSOL, all compatible with WSDL version 1.1: WSOL version 1.0 [Pat03a], WSOL version 1.1 [Pat03b], and WSOL version 1.2 [Tos04d]. The newest version contains additions and modifications to the XML schemas for WSOL that I coded without Kruti Patel's help. In this Ph.D. dissertation, I discuss WSOL version 1.2. The compatibility of WSOL with WSDL version 2.0 [Chi03b] requires only relatively minor modifications and is planned for a future WSOL version.

In parallel with WSOL, several languages with partially overlapping goals appeared recently, but they do not support the concept of a class of service. I will explain in Section 3.6 that, compared to these related works, WSOL has several unique characteristics that can also be useful for the future development of Web Service technologies.

3.3 Concepts in WSOL

Now that I gave an overview of WSOL and explained my goals and requirements for its development, let me explain the constructs that I built into WSOL to satisfy these goals and requirements. The main **categories of constructs** in WSOL are:

1. service offering (SO),
2. constraint,
3. management statement,
4. reusability element, and
5. service offerings dynamic relationship (SODR).

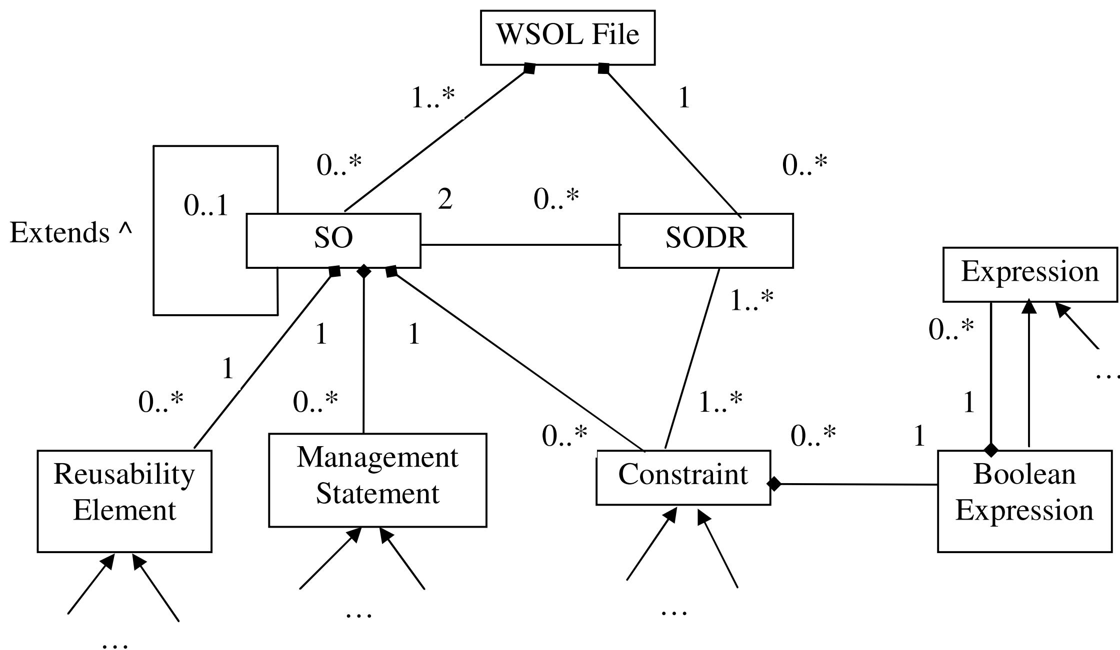


Figure 3.2 Partial UML Class Diagram for WSOL Concepts

Figure 3.2 is a UML class diagram illustrating the main WSOL constructs and their relationships. For example, this figure shows that WSOL files can contain zero or more service offerings (SOs) and/or zero or more service offerings dynamic relationships (SODRs). The figure also shows that while a definition of a service offering can span several WSOL files, a service offering dynamic relationship currently must be defined in only one WSOL file. To avoid overcrowding, I did not depict that constraints, statements, and reusability elements can also be specified inside WSOL files, but outside particular service offerings. In addition, I did not list all supported types of constraint, statement, reusability element, and expression.

I summarize the main characteristics of these categories of constructs and their relationships in the following five subsections. Further information can be found in other publications about WSOL and WSOI. The main WSOL publications are [Tos04b,

Tos03b]. Precise syntax (XML schemas) and illustrative examples of the WSOL language constructs are given in [Pat03b, Pat03a]. Several publications, such as [Tos03a, Tos02b], discuss additional WSOL topics that are only briefly mentioned in this Ph.D. dissertation.

3.3.1 Service Offerings

The main concept in WSOL is a **service offering (SO)**. Apart from the formal description of constraints and statements, a WSOL service offering can contain reusability elements. I will occasionally use the term '**service offering item**' to refer to a constraint, statement, or a reusability element inside a service offering.

In most cases, one service offering is defined for one particular Web Service. However, it is also possible to define in WSOL reusable service offerings that can apply to several Web Services, such as Web Services implementing same port types.

Figure 3.3 shows an example WSOL service offering *SO2* for the *buyStockService* Web Service. *SO2* extends another service offering *SO1*, includes the QoS constraint *QoScons1* defined outside any service offering, and contains the QoS constraint *QoScons2*, the pay-per-use price statement *Price1*, and the management responsibility statement *MangResp1*. WSOL definitions of service offerings are usually long and complex. Therefore, I show in this figure only example parts of service offering *SO2* that will be referenced in this section. Other constraints, statements, and reusability elements are left out for brevity. Further examples can be found in [Pat03b, Pat03a, Tos03a, Tos02r] and other publications about WSOL.

```
<wsol:WSOLdefinitions …
 xmlns:buyStock = ” http://www.buyStockPro.ca/buyStockFile.wsdl ” … >
 …
 <wsol:serviceOffering name = ” SO2 ”
  service = ” buyStock: buyStockService ”
  extends = ” tns: SO1 ”
  accountingParty = ” WSOL-SUPPLIERWS ”
  autoManipulation = ” True ” >
   <wsol:include constructName = ” tns: QoScons1 ”
    resService = ” buyStock: buyStockService ”
    resPortOrPortType = ” WSOL-EVERY ”resOperation = ”WSOL-EVERY ”
    resName = ” QoScons1buyStock ” />
   <wsol:constraint xsi:type = ” qosSchema: qosConstraint ”
    name = ” QoScons2 ”
    service = ” WSOL-ANY ”
    portOrPortType = ” WSOL-EVERY ” operation = ” WSOL-EVERY ” >
     <expressionSchema:booleanExpression >
      <expressionSchema:arithmeticExpression >
       <expressionSchema:QoSmetric
        metricType = “ QoSMetricOntology: ResponseTime ”
        service = ” WSOL-ANY ”
        portOrPortType = ” WSOL-ANY ” operation = ” WSOL-ANY ”
        measuredBy = ” WSOL_INTERNAL ” />
       </expressionSchema:arithmeticExpression>
       <expressionSchema:arithmeticComparator type= ” &lt; ” />
       <expressionSchema:arithmeticExpression >
         <wsol:numberWithUnitConstant>
           <wsol:value>0.3</wsol:value>
           <wsol:unit type = ” QoSMeasOntology: second ” />
         </wsol:numberWithUnitConstant>
       </expressionSchema:arithmeticExpression>
     </expressionSchema:booleanExpression>
   </wsol:constraint>
   <wsol:statement xsi:type= ” priceSchema: priceStatement”
    name = ” Price1 ”
    service = ” buyStock: buyStockService ”
    portOrPortType = ” buyStock: buyStockPort ”
    operation = ” buyStock: buySingleStockOperation ” >
     <wsol:numberWithUnitConstant>
       <wsol:value>0.01</wsol:value>
       <wsol:unit type = ” currencyOntology: CanadianDollar ” />
     <wsol:numberWithUnitConstant>
   </wsol:price>
   …
   <wsol:statement xsi:type= ” managementResponsibilitySchema:
    managementResponsibility ” name = ” MangResp1 ” >
     <wsol:supplierResponsibility scope = ” tns: AccRght1 ” />
     <wsol:consumerResponsibility scope = ” tns: Precond3 ” />
     <wsol:independentResponsibility scope = ” tns: QoScons2 ”
      entity = ” http://www.someThirdParty.com ” />
   </wsol:managementResponsibility>
</wsol:serviceOffering>
</wsol:WSOLdefinitions>
```

Figure 3.3 Parts of an Example WSOL Service Offering

Important management information is conveyed in the attributes of the WSOL <*serviceOffering*> element. The '*service*' applicability domain attribute specifies for which Web Service the service offering should be used. The value of this attribute is a qualified name whose namespace part refers to a WSDL file, while its local part refers to the name of a Web Service defined in that WSDL file. In Figure 3.3, the service offering '*SO2*' is defined for the '*buyStockService*' Web Service described in the WSDL file '*http://www.buyStockPro.ca/buyStockFile.wsdl*'. For the majority of constraints, statements, reusability elements, and QoS metrics used inside a service offering, the applicability domain is determined with three attributes: '*service*', '*portOrPortType*', and '*operation*', according to the rules presented in [Tos03a, Pat03a].

The '*extends*' attribute specifies the name of a previously defined service offering that the new service offering extends through single inheritance. In Figure 3.3, the service offering '*SO2*' extends some service offering '*SO1*' defined in the same WSOL file.

Due to the special purpose and characteristics of accounting parties discussed in Section 2.3, the special attribute '*accountingParty*' of the WSOL <*serviceOffering*> element determines the accounting party for a service offering. While other management parties are specified in management responsibility statements discussed in Subsection 3.3.3, accounting parties are not specified in such a way. This solution ensures that every service offering has exactly one accounting party. In Figure 3.3, the accounting party for '*SO2*' is the provider (supplier) Web Service.

The '*autoManipulation*' attribute determines whether the provider is allowed to perform switching from this WSOL service offering to another service offering without pre-

viously asking the consumer for explicit confirmation. In the example shown in Figure 3.3, the value of this attribute is '*True*', which means that provider need not ask for consumer confirmation before switching. However, the provider has to inform the consumer after the switching. This and other aspects of switching between service offerings will be explained in Subsections 5.2.1 and 5.2.2.

There are several other optional attributes of the <*serviceOffering*> element [Pat03b], not illustrated in Figure 3.3. They are used to specify subscription duration, subscription price, duration of the validity period, or expiration time of the service offering.

A WSOL definition of a service offering can be also viewed as a reusability construct. Reusability is achieved through reusability elements and attributes discussed in Subsection 3.3.4 and in [Tos03a]. One example is the '*extends*' attribute. In addition, some service offerings (e.g., those containing only QoS constraints) can be defined to apply to any Web Service by providing the special value '*WSOL-ANY*' for the '*service*' applicability domain attribute. Such reusable service offerings can then be applied to a particular Web Service using the WSOL inclusion reusability element discussed in Subsection 3.3.4. In this way, different Web Services can provide the same service offering.

3.3.2 Constraints and Expressions

To perform management activities, it is important to distinguish between valid or expected situations and invalid or unexpected situations. As I mentioned in Section 2.3, various requirements and guarantees are specified in software engineering in the form of constraints. Constraints describe valid or expected situations related to provisioning and consumption of Web Services in a way that can be used for automated monitoring

[Tos04b]. As I will discuss in Section 3.4, different types of constraint can be used in different management areas. Consequently, I built into WSOL formal specification of several categories of constraints and, for most categories, several types of constraint within a category.

In WSOL, every **constraint** formally states some condition to be evaluated. The constraints can be evaluated before and/or after invocation of some operations or periodically, at particular date/time instances. Every constraint contains a **Boolean expression** that formally represents the condition to be evaluated.

WSOL enables the formal specification of:

1. functional (behavioral) constraints,
2. QoS (non-functional, extra-functional) constraints, and
3. access rights.

Functional (behavioral) constraints define conditions that a functionally correct (valid) operation invocation must satisfy. They usually check some characteristics of message parts of the invoked operation. WSOL enables specification of pre-, post-, and future-conditions, as well as invariants. Preconditions check that the consumer-provided inputs into the invoked operation and/or the initial state of the provider are valid. Postconditions check that the results provided to the consumer and/or the final state of the provider are valid. Invariants describe conditions that have to be satisfied both before and after the execution of the invoked operation. An invariant can be represented with a pair of a precondition and a postcondition, but explicit support for invariants has also been developed for WSOL. The novel concept of a future-condition [Tos02r, Pat03a] is introduced to model conditions evaluated some time after the provider finishes execution of

the requested operation and sends results to the consumer. It enables specification of operation effects that cannot be easily expressed with post-conditions. An example is delivery confirmation for goods bought using Web Services. I also studied the concept of a periodic future-condition, but it is not yet a part of WSOL because I estimated that potential uses are relatively rare, while the implementation is non-trivial.

QoS (non-functional, extra-functional) constraints check whether the monitored QoS metrics are within specified limits. QoS metrics describe properties such as performance, availability, or reliability. Many QoS constraints are evaluated for a particular operation invocation. Periodic QoS constraints are a special type of QoS constraints because they are evaluated periodically, at specified times, and independently from particular operation invocations. This is useful for checking QoS metrics such as average response time, throughput, and availability. QoS constraints usually describe QoS guarantees. However, QoS constraints for output-input operations and some periodic QoS constraints can be used for the specification of QoS requirements. WSOL QoS constraints contain specifications of what QoS metrics are monitored, as well as when and by what entity. However, definition of QoS metrics (i.e., how they are measured or computed) is done in external reusable and extensible ontologies. I will discuss this topic in Subsection 3.3.4.

The overhead of measurement and calculation of QoS metrics and evaluation of QoS constraints for every operation invocation can be too high for some circumstances. This overhead is lower for periodic QoS constraints. To reduce the run-time overhead of monitoring activities, at the cost of quantity and precision of management information, WSOL also enables occasional evaluation of QoS constraints. This means that WSOL constraints

checked before and/or after operation invocations can be evaluated occasionally, only for some randomly chosen invocations (on average: 1 in n). This is specified with the optional attribute '*evalPeriod*' of QoS constraints. For example, when this attribute is set to 5, this means that this QoS constraint is checked, on average, for one operation invocation out of five. To maximize usefulness of such constraint evaluations, the provider (and, potentially, the consumer) must not know in advance for which operation invocation the QoS metrics will be measured and/or the constraint will be evaluated. This is achieved when an independent accounting party randomly chooses between operation invocations and monitoring (i.e., measurements and evaluations) is performed only by third parties. To make up for cases when unsatisfied constraints are not checked, monetary penalties for not meeting such constraints should be relatively high.

A WSOL **access right** specifies conditions under which any consumer using the current service offering has the right to invoke a particular operation. For example, an access right can limit the time of day when an operation can be invoked. It can also limit the number and/or frequency of invocations. Access rights are used in WSOL for differentiation of service (i.e., the provided functionality). On the other hand, the specification of conditions under which a particular consumer or a class of consumers may use a service offering and other security issues are outside the scope of WSOL. While I recognize that security of computing systems is an extremely important issue, my Ph.D. research contributions, including the WSOL concept of an access right, are not about security of Web Services. WSOL access rights might be used as a very small part of comprehensive security solution for Web Services, but this is not why I integrated them into WSOL.

Note that in the above discussion I used the traditional distinction between functional and extra-functional constraints present in the software engineering literature, e.g., in [Beug99, Crn02, Beus00]. While this distinction is disputed by some, it is given for classification purposes. It is not reflected in the WSOL language because WSOL directly supports individual types of constraint: pre-condition, post-condition, invariant, future-condition, QoS constraint, periodic QoS constraint, and access right.

WSOL constraints are defined using the <*constraint*> element, which is independent of particular categories and types of constraints. The '*type*' attribute of the <*constraint*> element refers to the XML schema defining a particular type of constraint. Under my guidance, Kruti Patel has defined XML schemas for the above-mentioned types of constraint outside the XML schema for the WSOL language. This separation enables that, using the XML Schema mechanisms, the XML schemas for additional types of constraint can be defined and used without any modification of the XML schema for the WSOL language. If desired, the existing constraint schemas can also be removed or replaced without any modification of the WSOL language schema. For example, WSOL version 1.1 [Pat03b] significantly extended and improved specification of periodic QoS constraints, future-conditions, and (to a lesser degree) non-periodic QoS constraints, compared to WSOL version 1.0. However, these improvements did not require changes in the core WSOL language schema.

By default, consumers are responsible for meeting preconditions and access rights and providers for meeting the other constraints. The responsible party pays the penalty if the constraint is not met. However, providers can be responsible for some preconditions and consumers can be responsible for some QoS constraints expressing QoS requirements of

the provider. While the latter cases are relatively rare, WSOL version 1.2 [Tos04d] contains the '*supplierResponsibility*' attribute of the <*constraint*> element to enable specification whether the consumer or the provider is responsible for meeting a constraint. Further, new attributes '*periodic*', '*beforeOperation*', and '*afterOperation*' of the <*constraint*> element specify when a constraint should be evaluated. Default values for these attributes are built into XML schemas for particular types of constraint.

Boolean expressions in WSOL constraints can contain standard Boolean operators (AND, OR, NOT, IMPLIES, EQUIVALENT), references to operation message parts of type Boolean or to used QoS metrics of type Boolean or to other Boolean expressions, and comparisons of arithmetic, string, date/time, or duration expressions. WSOL also supports checking operation message parts that are arrays of any data type using quantifiers '*ForAll*' and '*Exists*'. Arithmetic expressions can contain standard arithmetic operators (+, -, unary -, *, /, **), arithmetic constants, and references to operation message parts of numeric data types or to used QoS metrics of numeric data types or to WSOL expressions of numeric data types. WSOL provides only basic built-in support for string and date/time/duration expressions.

In addition, it is possible to perform operation calls in any expression. The called (invoked) operation can be implemented by another Web Service, by the management party evaluating the given constraint, or by the Web Service for which the constraint is evaluated. In cases when the invoked operation is on the same management party that evaluates the expression, this operation is invoked using internal mechanisms, without any SOAP call, although the operation is described with WSDL.

When constraints are specified in programming languages, one of the guidelines is that their evaluation should not change the state of the module for which the constraint is evaluated [OMG03, Tos98]. WSOL operation calls to the Web Service for which the constraint is evaluated might violate this guideline. It is up to developers of WSOL service offerings to decide whether they want to follow this guideline and to enforce it. Since operation calls can have side effects, WSOL is not a pure expression language.

Referring to the WSOL example shown in Figure 3.3, the QoS constraint '*QoScons2*' contains a comparison of a measured QoS metric '*ResponseTime*' and the constant '*0.3 second*'. The values for the applicability domain attributes '*service*', '*portOrPortType*', and '*operation*' indicate that this QoS constraint is evaluated for every operation of the '*buyStockService*' Web Service, because this is the Web Service for which the containing service offering '*SO2*' is evaluated. Further explanation of WSOL constants such as '*WSOL-ANY*' and '*WSOL-EVERY*' can be found in [Tos03a, Pat03a]. The value '*WSOL_INTERNAL*' for the '*measuredBy*' attribute of the QoS metric states that the entity that evaluates this QoS constraint also measures the response time QoS metric.

Let me reflect a bit more on the decision to include the specification of functional constraints into WSOL, instead of developing a WSDL extension for their specification in WSDL files. While QoS constraints provide differentiation of QoS, functional constraints and access rights provide differentiation of service. I explained in Section 1.5 that one of my secondary goals for WSOL was to enable unified specification of various management information for Web Services in one language, instead of several separate languages. I particularly emphasize that WSOL enables association of management responsibility and monetary penalties with functional constraints. Also, while in most cases one

Web Service offers the same functional constraints to all consumers and these functional constraints do not change during run-time, in some cases a Web Service can offer several sets of functional constraints to consumers [Beug99]. For example, such a differentiation of service can be used for consumers executing on different devices.

3.3.3 Management Statements

While the explicit and formal specification of constraints supports various management activities, it is not enough for management of Web Services and their compositions. For some important categories of management information, such as prices and management responsibilities, the representation in the form of constraints containing Boolean expressions is not natural. In a way, constraints are prescriptive, while these additional categories of management information are descriptive. For example, prices are important for accounting management. Some of this additional information can be represented through attributes of WSOL elements for service offerings and constraints. In Subsection 3.3.1, I mentioned attributes of the *<serviceOffering>* element that are used to specify subscription price and subscription duration. However, using attributes is not always appropriate. For example, failure to satisfy the same constraint could result in different monetary penalties in different service offerings. Consequently, I decided to add into WSOL the concept of a (management) statement. Using statements instead of attributes provides easier differentiation and better reusability.

A WSOL **statement** is any construct, other than a constraint, that contains important management information about the represented class of service. For the formal specification of statements, WSOL contains the general *<statement>* element. It is analogous to

the general <*constraint*> element discussed in previous subsection. The '*type*' attribute of the <*statement*> element refers to the XML schema defining a particular type of statement. Under my direction, Kruti Patel has defined XML schemas for the following categories of management statement:

1. pay-per-use price statements,
2. subscription price statements,
3. monetary penalty statements, and
4. management responsibility statements.

Additional categories and types of management statement (e.g., policies) can be supported in WSOL by defining XML schemas for them.

A **pay-per-use price statement** defines the monetary amount a consumer using the particular service offering has to pay for invoking particular operation. Two types of pay-per-use price statements can be specified in WSOL. First, it is possible to specify price for a particular operation or a group of operations. Second, it is possible to specify a default price for operations of a particular Web Service. This default price applies to all operations for which pay-per-use price is not specified explicitly and only to those operations. In WSOL 1.2 [Tos04d], it is possible to designate that the recipient of the monetary amount in a pay-per-use price statement, subscription price statement, or a monetary penalty statement is a management third party. This represents the amount the consumer or the provider pays to the third party for its monitoring activities. It is also possible to explicitly specify whether the consumer or the provider performs the payment. If more than one price (e.g., a subscription price and a pay-per-use price, or two pay-per-use prices for the same operation) is specified for the same recipient, then they are added.

Another improvement in WSOL version 1.2, is the possibility of using expressions in pay-per-use price, subscription price, and monetary penalty statements. For example, this allows varying prices and penalties according to the time of a day. This also enables specification of parameterized statements in constraint group templates, discussed in Subsection 3.3.4. Further, such expressions can be used for indirect specification of pay-per-volume prices and various corrective/penalty actions.

Figure 3.3 shows an example WSOL pay-per-use price statement '*Price1*'. This statement describes that the price for using the operation '*buySingleStockOperation*' of the port '*buyStockPort*' of the Web Service '*buyStockService*' is 0.01 Canadian Dollar.

A **subscription price statement** defines the monetary amount a consumer or a provider has to pay for using a management third party during a particular period. As I mentioned in Subsection 3.3.1, subscription prices paid by consumers to providers can be specified through attributes of the <*serviceOffering*> element.

Apart from the pay-per-use and subscription payment models, other payment models are possible, such as the pay-per-volume model. Direct support for the pay-per-volume payment model, but can be added into a future version of WSOL.

Monetary penalty statements define the monetary amount that the provider Web Service has to pay to a consumer if the consumer invokes some operation and the Web Service does not fulfill all constraints it is responsible for. Three types of monetary penalty statements can be specified in WSOL. First, it is possible to specify penalty associated with a particular operation or a group of operations. If the provider does not fulfil all constraints associated with this operation for which it is responsible, then it pays this monetary penalty to the consumer. Second, it is possible to specify a default penalty for

operations of a particular Web Service. This default penalty applies to all operations for which monetary penalty is not specified explicitly and only to those operations. Third, it is possible to specify additional penalty associated with a particular constraint. Such penalty is paid as a surcharge to the penalty for not meeting all constraints associated with an operation. This enables associating higher penalty with more important constraints. For example, meeting a postcondition is often more important than meeting a QoS guarantee constraint and this can be associated with different monetary penalties. This type of monetary penalty statement is also used for constraints that consumers are responsible for, e.g., preconditions and access rights. The other two types do not apply to these constraints. If more than one monetary penalty is specified for the same operation, then they are added. Monetary units used in price and penalty statements are defined in external ontologies, which are examined in Subsection 3.3.4.

Specification of monetary penalties is not the only WSOL construct that describes what happens if some constraints were not met. The WSOL concept of a service offerings dynamic relationship (SODR) is also used for this purpose, although in a different way. This concept, described in Subsection 3.3.5, is used to specify what is the appropriate replacement service offering if a particular group of constraints was not met. As will be described in Section 4.2, this information can be used to change (in my terminology: 'switch') what service offering is used by the consumer.

I also looked at other possible mechanisms to specify what happens if some constraints were not met, such as invocation of some management operations or opening a trouble ticket. SLAs often specify modes of compensating service consumers in case service providers do not meet the guaranteed service levels. While additional penalty

mechanisms might be useful in intra-enterprise scenarios in which there is a WSCM entity and internal payments are not appropriate, I concentrated my research on business-to-business scenarios and made no assumptions about WSCM entities. Most consumers in business-to-business scenarios are not interested in knowing HOW the provider will meet the guarantees, but that they DO meet the guarantees. Opening a trouble ticket is of no benefit to a consumer if it is not followed by an appropriate corrective action. On the other hand, a monetary penalty is a direct compensation to a consumer and can be used for management activities on the provider side. For example, 'management by contract' [Sal04] also relies on prices and monetary penalties (and not other actions specified in contracts) to achieve business-driven management of IT (Information Technology) systems. Therefore, my impression is that the corrective/penalty mechanisms such as invocation of management operations and opening of a trouble ticket are important for manageability, but that they can be hidden from consumers. In many cases, they are implementation details of the provider Web Service. If they are logically parts of a Web Service composition, then they have to be specified in a description language for Web Service compositions (e.g., BPEL4WS), not in WSOL. Consequently, I decided not to build them into the current version of WSOL. Nevertheless, if the need for such mechanisms in WSOL is determined, they can be added later. The addition of these mechanisms into the WSOL language would not be too complex, because it would use the WSOL concept of an operation call and the Service Offering Management (SOM) port types discussed in Section 4.3. However, the implementation of these mechanisms in the corresponding management infrastructure would not be trivial.

A **management responsibility statement** specifies what entity has the responsibility for checking a particular constraint, a constraint group, or the complete service offering. As discussed in Section 1.3, a management entity can be the provider Web Service, the consumer, or an independent third party trusted by both the provider and the consumer. Management third parties specified in management responsibility statements act as SOAP intermediaries. Due to the specifics of accounting parties, it is not possible to use a management responsibility statement to specify the accounting party for a service offering. Instead, the '*accountingParty*' attribute of the <*serviceOffering*> element has to be used. Figure 3.3 also shows an example WSOL management responsibility statement, '*MangResp1*'. In this example, the provider (supplier) Web Service evaluates the access right '*AccRght1*', the consumer evaluates the pre-condition '*Precond3*', and the specified third (independent) party evaluates the QoS constraint '*QoScons2*'.

3.3.4 Reusability Elements and Attributes

I mentioned in Section 1.5 that one of my secondary goals for developing WSOL was to enable reusable specification of service offerings. Therefore, I built into WSOL expressive reusability constructs to enable easier specification of new service offerings from existing service offerings of the same Web Service or other Web Services, as well as easier comparison of different service offerings in the process of selection of Web Services and service offerings. Reusability constructs in WSOL were discussed in detail in [Tos03a]. Due to space limits, I present only the main contributions in this Ph.D. dissertation, while the details and illustrative examples can be found in [Tos03a].

In WSOL, three **categories of reusability constructs** can be identified:

- definition of a service offering,
- reusability elements, and
- reusability attributes.

I already discussed in Subsection 3.3.1 how definitions of service offerings can be viewed as reusability constructs. Reusability elements are separate XML elements in the WSOL grammar, while reusability attributes are XML attributes of service offerings, constraints, and management statements.

WSOL files can contain several special **reusability elements**:

1. definition of a constraint group (CG);
2. inclusion of a previously defined constraint, statement, or constraint group;
3. definition of a constraint group template (CGT);
4. instantiation of a constraint group template;
5. reference to an expression; and
6. declaration of an operation call.

Constraint group

A **constraint group (CG)** is a named set of service offering items: constraints, statements, and reusability elements. While constraints specified directly inside a service offerings have all to be satisfied, it is possible to specify constraints groups in which all, at least one, or exactly one of the constraints have to be satisfied [Pat03b]. However, all statements and reusability elements are applied in every constraint group, even when only some constraints have to be satisfied. The names 'service offering item group' or 'item group' might have been more suitable for this WSOL construct. However, I kept the name 'constraint group' for backward compatibility with early versions of WSOL where

constraint groups contained only constraints and to show that constraints have special status in constraint groups.

The WSOL concept of a constraint group has several benefits. First, a constraint group can be reused across service offerings and even across Web Services as a unit, using a single WSOL inclusion reusability element discussed later in this subsection. Second, it is possible to specify in a single management responsibility statement that all constraints from a constraint group are evaluated by the same management entity. Third, constraints and statements (as well as nested constraint groups) in different constraint groups can have the same relative name, but different absolute names. A relative name of a WSOL constraint, statement, or constraint group is a string that must be unique inside the containing service offering, constraint group, or constraint group template. An absolute name of the same WSOL item is a namespace-qualified string that contains the relative name of this item, the relative names of all containing service offerings, constraint groups, and constraint group templates, and the namespace of the most general of these relative names. For example, if a service offering *SO7* contains a constraint group *CG3* that contains a precondition with the relative name *Precond5*, then the absolute name of *Precond5* is *SO7.CG3.Precond5*. Consequently, using constraint groups enables reuse of relative names. Fourth, constraint groups can be used for different logical groupings of constraints and statements. For example, one can use a constraint group to group constraints and statements related to a particular port, port type, or operation. In this way, the concept of a constraint group complements the concept of a service offering that assembles constraints and statements on the level of a Web Service. One can also use constraint groups to define aspects of service offerings. For example, one can group all functional

constraints for a Web Service into one constraint group, QoS constraints for the same Web Service into another constraint group, and access rights for this Web Service into a third constraint group. This supports separation of concerns.

The syntax and semantic similarities and differences between constraint groups and service offerings were examined in [Tos03a]. Syntactically, a WSOL definition of a constraint group is similar to a definition of a service offering. It is a set of defined or included constraints, statements, and reusability constructs that all refer to the same Web Service. WSOL supports extension of constraint groups (and constraint group templates, explained later in this subsection), similarly to the extension of service offerings. However, arbitrary levels of nesting of constraint groups are allowed, while service offerings must not be nested. Another syntactic difference is that the <*serviceOffering*> element has several management attributes that cannot be specified for constraint groups. These attributes specify the accounting party, whether automatic adaptation of service offerings is allowed, subscription duration, subscription price, duration of the validity period, and expiration time of the service offering. By 'automatic' adaptation, I mean spontaneous adaptation (e.g., switching) that is not explicitly requested from outside. These attributes reflect special run-time characteristics of service offerings. Most importantly, consumers can choose and use service offerings, not constraint groups. Another consequence of this semantic difference is that service offerings dynamic relationships (SODRs), discussed in Subsection 3.3.5, can be specified only for service offerings, not for constraint groups.

Inclusion element

The **inclusion element** enables reusing constraints, statements, and constraint groups that have been already defined elsewhere. For example, when a new service offering is

defined and some of the contained constraints, statements, and constraint groups have been already defined elsewhere, there is no need to define them again. They can simply be included into the new containing service offering using the WSOL inclusion element. Consequently, reusable constraints, statements, and constraint groups can be defined only once and then included in many different service offerings, constraint groups, and/or constraint group templates, even across Web Services. Reusable constraints, statements, and constraint groups can be defined inside service offerings, but also outside any service offering, e.g., in libraries of reusable WSOL items. Using the inclusion element, the relative name of the included constraint, statement, or constraint group can be changed and its applicability domain can be made more specific. The new relative name is specified with the attribute '*resName*'. The new applicability domain is specified with the attributes '*resService*', '*resPortOrPortType*', and '*resOperation*'. In this way, one constraint, statement, or constraint group can be included in the same service offering several times, to apply to different domains.

The WSOL inclusion reusability element is illustrated in Figure 3.3, where the QoS constraint '*QoScons1*' is included into the service offering *SO2* to apply to every operation of every port of the '*buyStockService*' Web Service. The QoS constraint '*QoScons1*' was previously defined somewhere in the same file, but outside any service offering. After the inclusion, the relative name of the included constraint is '*QoScons1buyStock*'.

The addition of naming and referencing of expressions into WSOL [Pat03b] enables referencing an existing named expression inside another expression. When a WSOL file is parsed, the reference is replaced with the complete referenced expression. This complements the reusability that can be achieved through the inclusion element.

Constraint group template

A **constraint group template (CGT)** is a parameterized constraint group. The concept of a constraint group template in WSOL is a very useful and powerful specification mechanism. Many classes of service contain constraints with the same structure, but with different numerical constant values. In my opinion, it is the most useful reusability construct in WSOL. As will be discussed in Section 3.6.1, some languages for the formal specification of SLAs for Web Services, most notably IBM's WSLA [Kel03, Lud03], also use templates as a reusability construct. WSOL contains two constructs for constraint group templates – one for their definition and the other for their instantiation.

At the beginning of a **definition of a constraint group template**, one declares one or more abstract constraint group template parameters, each of which has a name and a data type. Constraint group template parameters often have the data type '*numberWithUnit*', which requires additional information about the used measurement unit. The declaration of parameters is followed by the definition or inclusion of constraints, statements, and reusability elements, in the same way as for constraint groups. Constraints and statements inside a constraint group template can contain expressions with constraint group template parameters. Constraint group templates can be only defined outside service offerings.

In an **instantiation of a constraint group template**, concrete values are supplied as substitutions for all constraint group template parameters. One constraint group template can be instantiated many times with different parameter values, in different service offerings and for different Web Services. The result of every such instantiation is a new constraint group. The applicability domain of this new constraint group can be made more specific than the applicability domain of the original constraint group template.

Declaration of an operation call

In Subsection 3.3.2, I mentioned that WSOL expressions can contain operation calls to another Web Service, to the management entity evaluating the given constraint, or to the Web Service for which the constraint is evaluated. A **declaration of an operation call** is used to enable referencing results of the same operation call (i.e., invocation) in different sub-expressions of one constraint or in several related constraints, without re-invoking the operation.

Reusability attributes

In addition to the discussed XML reusability elements, several XML **reusability attributes** are built into the WSOL grammar:

1. extension (single inheritance) of service offerings, constraint groups, and constraint group templates;
2. definition of applicability domains;
3. specialization of applicability domains during inclusion and constraint group template instantiation;
4. declaration of use of QoS metrics, measurement units, and monetary units defined in external ontologies.

Most importantly, the '*extends*' attribute can be used to specify extension (single inheritance) of service offerings, constraint groups, and constraint group templates. When a new service offering is defined as an extension of an existing service offering, the extending service offering contains all service offering items (constraints, statements, and reusability elements) as the extended service offerings and can define some additional service offering items. In this way, new similar service offerings, constraint groups, and con-

straint group templates can be defined relatively easily from existing ones. I also studied the use of multiple inheritance in WSOL, but decided not to support it due to complexities, such as various potential conflicts. However, similar effects can be achieved by including multiple constraint groups inside a new one [Tos03a, Tos03b].

An **applicability domain** defines to which operations, port types, ports, and/or Web Services the given WSOL construct applies. It is specified in applicability domain attributes '*operation*', '*portOrPortType*', and/or '*service*'. Every WSOL service offering, constraint, statement, constraint group, constraint group template, and declaration of an operation call is defined for some applicability domain. Declarations of used QoS metrics, discussed later in this subsection, also contain specification of applicability domains, denoting for which operations, port types, ports, and/or Web Services the QoS metric is measured or computed.

In WSOL, applicability domains can be abstract or specific. A specific applicability domain refers to a particular operation (or a group of operations) of a particular port (or a group of ports) of a particular Web Service. For example, the applicability domain of the QoS constraint '*QoScons2*' shown in Figure 3.3 is abstract – every operation of every port of any Web Service. In an abstract applicability domain, the value of one or more of the applicability domain attributes refers to any operation, port type, port, and/or Web Service. For example, the applicability domain of the pay-per-use price statement *Price1* in the same figure is specific – the operation '*buySingleStockOperation*' of the port '*buyStockPort*' of the Web Service '*buyStockService*'. Abstract applicability domains is beneficial for reusability, while specific applicability domains are necessary for manageability, particularly precise monitoring of Web Services [Tos03a]. To ensure manageabil-

ity of WSOL items, and at the same time provide some support for reusability, I developed and integrated into WSOL several original solutions. These are: special built-in constants '*WSOL-ANY*', '*WSOL-EVERY*', '*WSOL-MANY*', and '*WSOL-ALL*' for applicability domain attributes, the concept of an actual applicability domain, the concept of a subdomain, a set of rules for relationships between applicability domains of containing and contained WSOL constructs, and support for specialization of applicability domains. I described, discussed, and illustrated them with examples in the extended version of [Tos03a]. In this Ph.D. dissertation, I only briefly present specialization of applicability domains.

As I mentioned, when a service offering, constraint, statement, or constraint group is included and when a constraint group template is instantiated, its abstract applicability domain can be made more specific. In other words, some or all applicability domain attribute values that were '*WSOL-ANY*' can be substituted with names of particular operations, port types, ports, and/or Web Services or with appropriate WSOL constants. This is expressed in the attributes '*resService*', '*resPortOrPortType*', and '*resOperation*' of WSOL elements for inclusion and for instantiation of constraint group templates.

An important reusability feature of WSOL is that QoS metrics, measurement units, and monetary units used in WSOL specifications are not actually defined in WSOL files. For the specification of QoS constraints, WSOL uses external ontologies of QoS metrics and measurement units, while for the specification of prices and monetary penalties it uses ontologies of monetary (currency) units. Such ontologies contain precise definitions of how the QoS metrics are measured and/or calculated and how the QoS metrics, measurement units, or monetary units relate to each other. These external ontological defini-

tions can be reused for different Web Services. The outsourcing of definitions of QoS metrics, measurement units, and monetary units can have several other benefits, discussed in [Tos03a]. First, such ontologies can be extended without any modification of WSOL files that use them. Second, the use of external ontologies eases comparisons between WSOL service offerings specified for the same or for different Web Services. Third, the use of common external ontologies can decrease chance of semantic misunderstanding between provider Web Services and their consumers. Fourth, the use of such external ontologies might ease compilation of WSOL files and reduce run-time overhead of their monitoring, metering, and evaluation.

I summarized requirements for such ontologies in [Tos02b] and discussed the related WSOL topics in [Tos03a]. The development of the described ontologies is a separate research issue from the development of WSOL. While it is an important issue, it was not a priority for my work on WSOL. Ideally, appropriate standardization bodies would develop such ontologies and make them well-known. Such standardization is useful because existence of multiple ontologies for the same concepts reduces interoperability. However, since a standardization process is long and tedious, it is practical to allow other interested parties to define and publish ontologies to be used until standards are finalized. For temporary use in the example WSOL files, Kruti Patel defined [Pat03a] very simple ontologies of QoS metrics as collections of names with information about appropriate data types and measurement units. Similarly, ontologies of measurement and monetary units are simple collections of names without any additional information. A more appropriate definition of ontologies of QoS metrics, measurement units, as well as monetary units is

planned for the future. Improvements in the definition of these ontologies will not modify the example WSOL files that use these ontologies.

Once a QoS metric is defined in a reusable external ontology, its use for particular operations, port types, ports, and/or Web Services is declared in WSOL files. This is done in the WSOL element <*QoSmetric*> for the **declaration of a used QoS metric**. This reusability element contains the attribute '*metricType*' to refer to the ontological definition of the QoS metric, the attribute '*measuredBy*' to refer to the management party that performs measurements and/or calculations, and applicability domain attributes. For example, the QoS constraint '*QoScons2*' in Figure 3.3 uses a response time QoS metric measured for the operation for which this constraint is evaluated. The same management party that evaluates '*QoScons2*' performs the measurement of this QoS metric. Some QoS metrics are measured, while some are calculated. Declarations of use of calculated QoS metrics have one or more <*usedQoSmetric*> sub-elements, which declare QoS metrics used in the calculation.

3.3.5 Service Offerings Dynamic Relationships (SODRs)

An important topic for WSOL is how to represent relationships between service offerings. These relationships have to be specified for at least three purposes. The first one is to provide a more straightforward and more flexible specification of new service offerings. This is needed to specify relatively similar service offerings of one Web Service, as well as relatively similar service offerings of similar Web Services. The second purpose is to enable easier comparison, selection, and negotiation of service offerings. The third purpose is to support dynamic adaptation of Web Service compositions based on the ma-

nipulation of service offerings, which I will discuss in detail in Chapter 4. I studied various possibilities for the specification of relationships between service offerings and concluded that a difference should be made between static and dynamic relationships.

Static (development-time) **relationships between service offerings** show similarities and differences between service offerings that do not change during run-time. Three important examples of static relationships between service offerings are extension (single inheritance) of service offering, inclusion of the same constraint or statement, and instantiation of the same constraint group template with different parameter values. These relationships are crucial for easier and more flexible specification of new service offerings from existing ones. They can also be used for easier comparison, selection, and negotiation of service offerings. In WSOL, static relationships between service offerings are expressed with the discussed reusability constructs, inside definitions of service offerings.

Dynamic (run-time) **relationships between service offerings** are those that can change during run-time, e.g., after dynamic creation of a new class of service. For example, one such dynamic relationship can state what service offering could be an appropriate replacement if a particular constraint or a group of constraints from some other service offering were not met in the past and/or cannot be met in the future. Such dynamic relationships can change relatively frequently. It is advantageous when a change in a dynamic relationship does not affect definitions of the related service offerings. Consequently, dynamic relationships should not be specified inside definitions of service offerings. Both static and dynamic relationships between service offerings are useful for easier comparison, selection, and negotiation of service offerings. In addition, dynamic relation-

ships are crucial for the mechanisms for dynamic adaptation of Web Service compositions discussed in Chapter 4.

In WSOL, a dynamic relationship between service offerings is expressed with a special construct – a **service offerings dynamic relationship (SODR)**. Service offerings dynamic relationships are specified outside definitions of service offerings, often in separate files. A WSOL service offerings dynamic relationship can be specified as a triple <*SO1, S, SO2*> or as a triple <*SO1, E, SO2*> [Tos04b, Pat03b]. Here:

1. *SO1* is the used service offering,
2. a) *S* is the set of constraints from SO1 that are not satisfied; b) *E* is a Boolean expression relating unsatisfied constraints from SO1, and
3. *SO2* is the appropriate replacement service offering.

If the format <*SO1, S, SO2*> is used, then *SO2* is the appropriate replacement for *SO1* if all constraints from *S* are unsatisfied. I also looked at the possibility that the set *S* also contains constraint groups from *SO1* that are not satisfied. Since the implementation of the latter specification would introduce some complexities in the management infrastructure, I decided that WSOL 1.1 should not support this specification.

If the format <*SO1, E, SO2*> is used, then *SO2* is the appropriate replacement for *SO1* if the expression *E* is true. The former format is simpler and easier to use during run time, so it was the only format for service offerings dynamic relationships in early versions of WSOL. In the new WSOL version 1.1, I added the latter format, which is more powerful and can be used instead of the older format.

```
<sodr:SerOffDynRel sodrName = " SODR1 " >
  <sodr:currentSO name = " serviceOfferingsFile: SO2 " />
  <sodr:unsatisfiedConstraints>
    <expressionSchema:booleanExpression>
      <expressionSchema:booleanExpression>
        <sodr:unsatisfiedConstraintRef
         name = " serviceOfferingsFile: SO2.QoScons1 " />
      </expressionSchema:booleanExpression>
      <expressionSchema:binaryBooleanOperator type = " OR " />
      <expressionSchema:booleanExpression>
        <sodr:unsatisfiedConstraintRef
         name = " serviceOfferingsFile: SO2.QoScons2 " />
      </expressionSchema:booleanExpression>
    </expressionSchema:booleanExpression>
  </sodr:unsatisfiedConstraints>
  <sodr:replacementSO name = " serviceOfferingsFile: SO5 " />
</sodr:SerOffDynRel>
```

Figure 3.4 An Example Service Offerings Dynamic Relationship (SODR)

Figure 3.4 shows an example service offerings dynamic relationship in the format *<SO1, E, SO2>*. It specifies that if any of the constraints '*QoScons1*' and '*QoScons2*' from some service offering '*SO2*' is not satisfied, then the appropriate replacement service offering is '*SO5*'. Both '*SO2*' and '*SO5*' are defined in some external file, pointed to by the '*serviceOfferingsFile*' namespace.

3.4 How WSOL Supports Management Activities

Appropriate specification of management information, such as WSOL constraints and management statements, is necessary for successful management activities. In this subsection, I will explain how describing a Web Service in WSOL, in addition to WSDL, is useful for both Web Service Management and Web Service Composition Management activities. As I will mention at the end of this subsection, WSOL can also be used in selection of Web Services and classes of service that are best for particular circumstances.

The crucial WSOL support for Web Service Management applications is the **formal specification of various types of constraint and management statement**, in a format that can be used for automatic generation of constraint-checking code [Tos04b]. WSOL describes for Web Services what QoS metrics to measure or calculate; what constraints (requirements and guarantees) to evaluate; when, where, and to some extent how to perform these monitoring activities; and what are the monetary consequences of meeting or not meeting the constraints. This management information is the basis for Web Service monitoring and control of a Web Service to meet its guarantees.

Different types of WSOL constraint and management statement are useful in different management areas. QoS constraints are particularly useful in performance management because they prescribe which performance-related QoS metrics to monitor, where and how to do this monitoring, how to calculate aggregate QoS metrics, and what are the expected values of QoS metrics. When a WSOL service offering is viewed as a simple SLA, a WSOL QoS constraint is a Service Level Objective (SLO). Functional constraints are beneficial in fault management, particularly to determine whether a Web Service behaves correctly. Access rights limit access to operations and ports of a Web Service. While WSOL access rights are used primarily for service differentiation, they could also be used as one very small part of a comprehensive security management solution for Web Services. Statements about pay-per-use prices and monetary penalties, along with the attributes specifying subscription price, are valuable in accounting management, particularly billing. Management responsibility statements, along with the attributes specifying accounting parities and applicability domains, can be used in configuring Web Services and their compositions. All WSOL constructs can be useful for configuration manage-

ment of Web Services and their compositions, particularly in selection of Web Services and their classes of service. WSOL constraints and statements can also be viewed as two categories of management policies because they are declarative descriptions of what must be accomplished by management.

In WSOL service offerings, **management third parties** and **accounting parties** are designated **explicitly**. A management third party that measures or calculates a QoS metric is indicated in the attribute '*measuredBy*' of the declaration of use of this QoS metric. A management third party that acts as SOAP intermediary is specified in management responsibility statements. A management party that acts as a probe is modeled as a separate Web Service that provides results of its measurements through operations of some agreed-upon port types. To transport the results to another management party, these operations can be invoked in WSOL QoS constraints using the WSOL operation call mechanism. The information about the probe is specified in an attribute of the operation call WSOL element. If the called operation is input-output, the management information is pulled from the probe. If the operation is output-only or output-input, the probe pushes management information to the party evaluating the containing constraint. Due to the special purpose and characteristics of accounting parties, WSOL has special support for their specification using the attribute '*accountingParty*' of the XML element for service offerings.

Other **attributes** of WSOL constructs are also relevant for management. The subscription and validity period attributes of service offerings are useful for accounting management. The applicability domain attributes determine for which Web Service, port, and operation a WSOL construct or a QoS metric should be monitored. As will be discussed

in Section 5.6, an attribute of QoS constraints can be used to indicate their occasional evaluation, while child elements of periodic QoS constraints are used to specify evaluation times and periods.

The mechanisms for dynamic adaptation of Web Service compositions based on the manipulation of service offerings, to be presented in Chapter 4, are useful for both Web Service Composition Management and Web Service Management. The crucial WSOL language support for such manipulation of service offerings is the **explicit specification of dynamic and static relationships between service offerings**. In particular, WSOL specification of service offerings dynamic relationships is essential for switching (particularly provider-initiated), deactivation, and reactivation of service offerings. WSOL reusability elements and attributes can be used for dynamic creation of new service offerings and, to a lesser extent, other dynamic adaptation mechanisms, e.g., consumer-initiated switching. In addition, WSOL supports dynamic manipulation of service offerings with the '*autoManipulation*' attribute of the <*serviceOffering*> element, which determines the order of steps in the switching of service offerings.

In addition to Web Service Management and Web Service Composition Management, WSOL can also be used for used for selecting provider and consumer Web Services and their classes of service that are best for particular circumstances. Selection of Web Services and their classes of service might be viewed as a configuration management activity for Web Service compositions. In an ideal case, it would be possible to perform such selection dynamically (e.g., as a part of dynamic composition of Web Services), without prior knowledge between the consumer and potential provider Web Services. The selection process can use various comparisons and/or negotiation. Because I focused my re-

search on management of Web Services and Web Service compositions, I had to leave research of topics related to comparison, negotiation, and selection for future work. However, let me briefly discuss potential applications of WSOL in this area.

As the number of Web Services that offer similar functionality (e.g., the same WSDL port types) increases in the global market, the offered QoS, the price, the price/performance ratio, and manageability will become the main competitive advantages. The comprehensive specification of Web Services and service offerings in WSOL supports selecting the most appropriate Web Services and service offerings. Consumers get additional flexibility to better choose service and QoS that they will receive and pay for and minimize thus the price/performance ratio and/or the total cost of received services. On the other hand, provider Web Services with multiple service offerings can accommodate more diverse consumers and execution circumstances. They have more flexibility in selecting consumers they will accept (and their service levels) to achieve maximal monetary gain with optimal utilization of resources. [Men01] noted that a comprehensive formal specification of services to be composed helps reduce unexpected (side-effect) dynamic interactions between them. Since WSOL provides such a description for Web Services, it can improve successfulness of selection, composition, and cooperation.

It is important to note that the process of comparison, negotiation, and selection of Web Services, their functionality, and particularly QoS is a very complex issue, without a simple and straightforward solution. Often, service offerings or other contracts cannot be easily compared as they can have many dimensions of difference. For example, when one service offering constrains availability and does not limit response time, while another service offering does the opposite, it is very complex to say which one is 'better', unless

precise priorities are established beforehand. Therefore, I suggest using static and, particularly, dynamic relationships between service offerings as guidelines for comparing service offerings. While the explicit WSOL specifications of static and dynamic relationships between service offerings can be used as guidelines for comparisons and easier selection of the most appropriate Web Services and their service offerings, these specifications are not enough. In addition, appropriate algorithms and heuristics have to be developed and this seems as a fruitful topic for future research. Integration of WSOL with Web Service discovery and selection technologies, such as UDDI, is another important topic that I had to leave for future research.

3.5 WSOL Tools

To verify the implementability of the WSOL syntax, Kruti Patel has developed a **WSOL parser** called 'Premier' [Pat03a, Pat03b]. Its implementation is based on the Apache Xerces XML Java parser. The inputs into this parser are WSOL files and WSOL-related files, such as WSDL files and ontology files. The parser produces DOM (Document Object Model) tree representations of the parsed files, as well as multiple symbol tables used for storing and retrieving data for semantic analysis. It detects and reports syntax errors and most semantic errors in WSOL files and, to a lesser extent, in WSOL-related files.

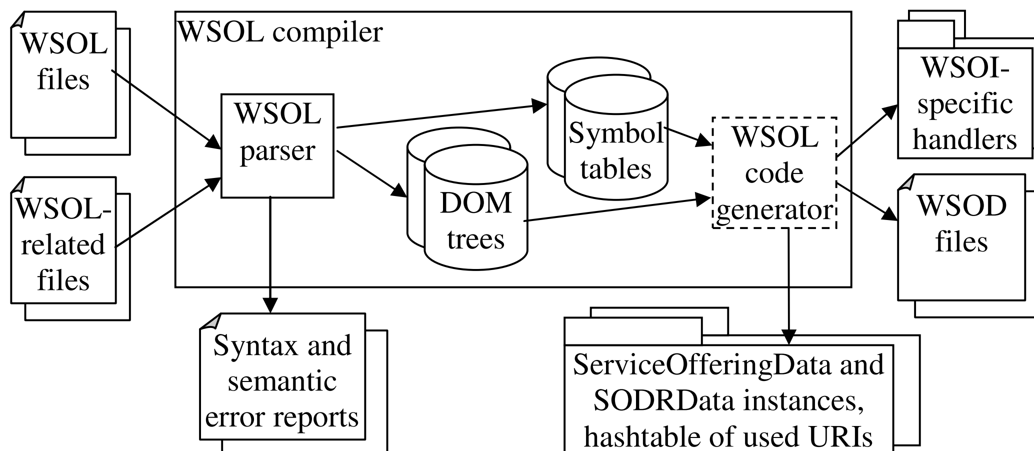


Figure 3.5 Compilation of WSOL Files

Under my instruction, Kruti Patel developed a case study, the '*buyStock*' Web Service, presented in [Pat03a, Pat03b]. For this case study, she wrote a series of WSOL examples to demonstrate usefulness of WSOL, soundness and implementability of the WSOL grammar, applicability of the 'Premier' WSOL parser, and correctness of its implementation. The examples demonstrated usefulness (pragmatics) of WSOL because they showed situations in which WSOL constructs enabled specification of beneficial information that could not be described in WSDL. They also tested syntax, and to some extent semantics, of all WSOL constructs because the 'Premier' WSOL parser processed correct examples or discovered deliberately introduced syntax and semantic errors.

In the future, the 'Premier' WSOL parser should be extended with a WSOL code generator to a full WSOL compiler. Figure 3.5 shows such a WSOL compiler and inputs, intermediate results, and final results of the compilation process. From DOM trees and symbol tables produced by a WSOL parser (e.g., 'Premier'), a WSOL code generator will automatically generate, without programmer intervention, modules that measure and/or

calculate QoS metrics, evaluate WSOL constraints, and perform accounting. In the monitoring and management Web Service Offerings Infrastructure (WSOI) I designed for WSOL, these modules are called 'WSOI-specific handlers'. Web Service Offering Descriptor (WSOD) [MaW04] files describe the ordering of WSOI-specific handlers and/or management third parties for particular circumstances. WSOI-specific handlers and WSOD files are explained in Section 5.3. A WSOL compiler will also produce appropriate instances of WSOI classes *ServiceOfferingData* and *SODRData*, as well as a hashtable of used URIs. *ServiceOfferingData* and *SODRData* store description of a service offering and a service offerings dynamic relationship, respectively. These and other WSOI data structures are presented in Section 5.5.

No WSOL code generator is implemented at this stage of WSOL-related research, because it was estimated that this implementation would bring little research benefits at the cost of significant effort. For the prototype implementation of WSOI [MaW04], Wei Ma, another Masters student working under my direction, manually implemented several specific WSOI-specific handlers. They are special cases of my general design and correspond to simple WSOL files from the '*buyStock*' case study. Wei Ma also wrote appropriate WSOD files for these WSOI-specific handlers.

The fact that the WSOL code generation was not implemented has a consequence that implementability of WSOL constructs was not fully proved. However, the 'Premier' WSOL parser and Kruti Patel's case study examples demonstrate parseability of WSOL constructs. The results of this parsing are standard internal data structures, analogous to internal data structures in many compilers. I cannot foresee any difficulties in code generation from such internal data structures. Further, the 'Premier' WSOL parser already

handles extension, instantiation of constraint group templates, and several other WSOL concepts. The focus of code generation would be WSOL expressions. In WSOL expressions, the most difficult to implement are the '*ForAll*' and '*Exists*' quantifiers used for array values. However, I intentionally kept these quantifiers simple, so that they can be implemented with one loop and appropriate helper variables. In this respect, WSOL expressions are simpler than OCL (Object Constraint Language), for which several compilers exist (an incomplete list is available at [Tov04]). Whenever I considered inclusion of a concept or feature into WSOL, I carefully studied how it supports the goals and requirements for my research and WSOL, how it relates to the other WSOL concepts and features, what code is needed for its compilation and run-time use, and experiences with similar concepts or features in other specification and/or programming languages. As a result, no concept or feature built into WSOL is problematic for compilation or run-time use. Note that ontological definitions of QoS metrics (and, thus, their implementability) are outside the scope of my work on WSOL. Nevertheless, if definitions of calculated QoS metrics use only WSOL expressions, I can claim with high confidence that they will be implementable. For the actual monitoring of measured QoS metrics, it is possible to develop a set of reusable code modules. For example, Wei Ma implemented several such modules for the prototype implementation of WSOI [MaW04]. It is also possible to use existing system management technologies—e.g., Application Response Measurement (ARM), Desktop Management Interface (DMI), Common Information Model (CIM), or Java Management Extensions (JMX) implementations—to collect values for measured QoS metrics. For example, [Deb03] discusses using CIM for monitoring of SLAs specified in WSLA. If a management party already supports such an instrumentation technol-

ogy and the results can be related to QoS metrics used in WSOL service offerings, a WSOL compiler would generate appropriate calls to this infrastructure. I left this issue for future research.

3.6 Comparisons of WSOL with Related Work

There is a considerable body of work in software engineering on the **formal representation of various types of constraint** and it significantly influenced my work on WSOL. Formal specification of functional constraints for programs has long history, although probably the best known result is the 'design by contract' [Mey92]. Several experimental QoS specification languages, often extensions of the CORBA IDL (Interface Definition Language) have been developed. Notable examples are QML (QoS Modeling Language) [Fro98, Fro99], QDL (QoS Description Languages) [Zin97], and QIDL [Bec99]. Specification and differentiation of access rights for subclasses and different classes of client of an object were suggested by [Hai90]. The work on WSOL was particularly influenced by **[Beug99]**, which elaborated the need for the unified formal specification of various types of constraint and comprehensive technical contracts for software components. [Dea99, Beus00, etc.] also emphasize the importance of formal specification of various types of constraint for static integration and maintenance of Commercial-Off-The-Shelf (COTS) software components. Specification of different types of constraint in XML was explored in **[Mck99]** and **[Jac00]**. The authors of [Mck99] specify in XML functional and simple QoS constraints, as well as authorization and obligation policies for entities in active networks. However, they do not address non-functional constraints in depth and specify authorization and obligation policies independently from entities they refer to. Similarly,

the authors of [Jac00] specify in XML functional constraints, some synchronization constraints, and basic invocation protocols for distributed objects. However, they do this in a very simple way. They do not address QoS constraints or access rights.

As I discussed in [Tos03b], the existing languages for the formal specification of constraints in software engineering could not have been applied directly to Web Services because they do not address well the specifics of the Web Services domain. They are not compatible with WSDL and all but [Mck99] and [Jac00] are not XML languages. They do not support well the heterogeneity of communication/interaction styles, the implementation independence, and/or the goal of dynamic, autonomous, business-to-business, and Internet-wide Web Service discovery, composition, and cooperation. None of these languages addresses multiple classes of service, which is the main goal of WSOL. Further, they do not enable specification of prices, monetary penalties, management responsibilities, and/or other management statements.

While at the time when I defined my Ph.D. research there was no work on the formal specification of classes of service, multiple types of constraints, and management statements for Web Services, several languages that partially address similar topics to WSOL appeared after I achieved results on WSOL. For example, some of these languages were developed for the formal XML specification of custom-made SLAs, policies, QoS descriptions, semantic descriptions, or other contracts for Web Services. While these related languages contain some alternatives to the solutions built into WSOL, none of them addresses the same set of issues as WSOL. While WSOL has limitations, it also has a set of important unique characteristics and advantages. While I developed most of my contributions independently from these languages, some of these languages occasionally influ-

enced my work on WSOL. For example, I independently envisioned management third parties acting as SOAP intermediaries, but only after reading [Lud02], I started to think how WSOL supports or could support management third parties acting as probes. In this section, I summarize some of these languages and emphasize the major similarities with and differences from WSOL. Kruti Patel's Masters thesis [Pat03a] contains another comparison of WSOL and the major related languages. It includes a tabular summary of similarities and differences, which she wrote under my instruction.

For easier comparisons with WSOL, I classify the related languages into languages for the formal specification of SLAs for Web Services, languages that have the concept of a class of service, languages intended for discovery and selection of Web Services, and other languages for additional description of Web Services. I think that the most important related languages to WSOL are languages for the formal XML-based specification of custom-made SLAs for Web Service, so I present them first. The other related works, even those with the concept of a class of service, are not as close to the goals of my Ph.D. research.

3.6.1 Related Languages for Custom-made SLAs for Web Services

The **Web Service Level Agreement (WSLA)** language [Kel03, Lud03, Lud02, Dan02] from International Business Machines (IBM) is the basis of the WSLA framework for the XML specification, monitoring, and management of custom-made SLAs. This general framework can be used for Web Services, but also for other electronic services. SLAs in this language contain three parts. The first part defines parties involved in the agreement – signatory parties (provider and consumer) and supporting (third) parties. For any party,

supported management operations can be defined. The second part contains one or more service definitions that contain information about monitored service objects. A ‘service object’ is an abstraction of relevant parts of the monitored service, e.g., a WSDL operation or a group of WSDL operations. SLA parameters are the monitored properties of a service object, e.g., response time measured by a particular party. Every SLA parameter is assigned one (QoS) metric, which defines how to measure and/or compute the value of the SLA parameters. An example of a metric is response time. One metric can be used by many SLA parameters, e.g., for different Web Services. The actual value of the metric is measured using a measurement directive or calculated using a function. A service object can also be associated with a number of schedules and triggers that describe when to perform monitoring and measurement actions. For reusability, metric macros can be defined. All these elements of a service description can be described in detail, achieving precise description of what QoS metrics are measured, where and how they are measured, as well as how to compute aggregate (composite) metrics from raw measured metrics. The third part describes obligations – service level objectives (SLOs) and action guarantees. An SLO describes guaranteed state for SLA parameters for particular period. It contains a logical (i.e., Boolean) expression to be evaluated and can contain specification of zero or more validity periods, evaluation schedules, and evaluation events. An action guarantee describes what is to be done (e.g., notification of a party or monetary payment) in particular situations. It contains a logical (Boolean) expression that is the precondition for the action, description of the action to be performed, and can contain specification of a limiting execution modality, evaluation schedules, or evaluation events. For reusability, obligation groups can be defined. The WSLA framework contains several tools for creation,

deployment, and compliance monitoring of SLAs, e.g., WSLA Compliance Monitor [Dan02].

Hewlett-Packard (HP) developed another language for the formal XML-based specification of SLAs for Web Services [Sah02a, Sah02b, Mac02]. [Sah02a] indirectly mentions that this language is a part of HP's **Web Service Management Language (WSML)**. There are no publications presenting other parts of WSML. For convenience, I will refer to this HP's SLA language as WSML. WSML is compatible with WSDL and now obsolete Web Services Flow Language (WSFL) [Ais02], which is one of the ancestors of the Business Process Execution Language for Web Services (BPEL4WS). An SLA in WSML consists of a description of the SLA's validity period, involved parties, and a set of SLOs. While the WSML schema presented in [Sah02a] does not support the concept of a management third party, a later publication [vMo02] discusses the use of management intermediaries in situations in which WSML is also used. Every SLO contains a description of days and times when the SLO is valid, as well as a set of clauses. A clause describes one or more items (e.g., operations) for which measurements are performed, times or events (e.g., operation completion) that trigger evaluation, measurement samples that are used for evaluation, one Boolean evaluation function, and an action that is performed if the evaluation function returns '*False*'. An evaluation function is applied over measurement samples to calculate a QoS metric and to evaluate an appropriate condition. It captures definition of this QoS metric. WSML does not prescribe evaluation functions that will be used – they can be defined in SLAs or in external libraries, using any appropriate mathematical XML-based language, such as MathML. This results in significant flexibility and extensibility. However, both consumer and provider have to additionally support the same

mathematical language to be able to use SLAs in WSML. In particular, the infrastructure for evaluation of WSML SLOs has to support this mathematical language and this is an additional overhead. In addition, this can lead to compatibility problems. The development of WSML was accompanied by the development of several tools [Sah02b, Mac02].

WSOL versus WSLA and WSML

I believe that WSLA and WSML are the main related languages to WSOL. These languages are high quality, mature, oriented towards management applications, accompanied by appropriate management infrastructures, and with industrial support. Since they have a number of similarities, I will now compare them both to WSOL. WSLA and WSML specify custom-made SLAs, but they do not contain an explicit concept of a class of service. SLAs in these two languages contain QoS guarantees in the form of SLOs and management information such as management responsibilities and prices. Since WSOL service offerings also specify QoS constraints (guarantees and requirements) and management information, they can be viewed as simple SLAs or other technical contracts. Let me outline some of the differences between WSOL service offerings and SLAs in WSLA and WSML. Custom-made SLAs explicitly specify information about consumers, which is implicitly assumed in WSOL. Further, SLAs in WSLA and WSML contain more detail for QoS guarantees than WSOL service offerings. For example, their description of evaluation schedules is more detailed than information provided in WSOL periodic QoS constraints. In addition, they can contain description of management actions to be performed if SLOs are not met, while WSOL only describes monetary penalties to be paid and assumes that the involved management parties will send appropriate notification messages. In these aspects, WSLA and WSML are more powerful than WSOL. WSLA

defines used QoS metrics within SLAs, while WSOL outsources such definitions to external reusable ontologies. While WSML can outsource definitions of evaluation functions to external libraries, its drawbacks are that it requires support for an additional language for mathematical expressions and that the choice of this language is open, leaving potential compatibility problems. It seems that all this results in higher run-time overhead for WSLA and WSML than the overhead of the simpler WSOL.

On the other hand, WSOL service offerings can contain formal specification of functional constraints, access rights, and other constraints and management statements that are not specified in SLAs. WSOL can be easily extended with the formal specification of additional types of constraint and management statement. Another advantage of WSOL is the broader set of powerful reusability constructs [Tos03a]. In particular, WSLA and WSML enable specification of templates, but only at the level of an SLA, not its parts. On the contrary, the WSOL concept of constraint group template can be used for complete service offerings or their parts. Further, WSLA and WSML do not support inheritance of SLAs nor inclusion of SLA parts defined elsewhere, while WSOL supports these and several additional reusability features. These WSOL reusability constructs make development and comparison of service offerings significantly easier. A very important concept present only in WSOL is a service offerings dynamic relationship (SODR).

While [Lud03] briefly mentions that WSLA can be used for the modeling of 'classes of service', they use this term in a different way from me. Their 'class of service' is specified within a custom-made SLA and refers to an obligation group. This means that a consumer cannot select such 'class of service' without negotiating the containing SLA. Further, the conditions under which such 'class of service' is in effect have to be speci-

fied for every contained obligation, since it is not possible to associate conditions with the whole obligation group. One of these conditions might be that a consumer chose a particular ‘class of service’. This can be specified in WSLA with a function call, in a relatively complex way. If a change of circumstances occurs and a ‘class of service’ can no longer be supported, either the conditions for every obligation have to describe that the ‘class of service’ becomes invalid or the whole SLA has to be renegotiated. On the other hand, a consumer chooses one WSOL service offering and there is no need for additional conditions inside a service offering to refer to consumer’s choice. A change of one service offering does not effect consumers using other service offerings. This means that while the [Lud03] notion of a ‘class of service’ can be used for service differentiation, my concept of a class of service is broader and more flexible.

Both WSLA and WSML are oriented towards management applications in inter-enterprise scenarios. They assume existence of some measurement and management infrastructure at both ends. Both languages are accompanied by appropriate management infrastructures that are more powerful, but also more complex, than WSOI, as discussed in Section 5.9. Recall that one of the secondary goals for my research was providing manageability with relatively low run-time overhead. Consequently, the simpler and more lightweight WSOL and WSOI might be a better choice for Web Services for which additional run-time overhead is an important issue and/or which execute in limited environments. Web Services in mobile, embedded, and ubiquitous computing are an example.

To summarize, WSLA and WSML are very good languages for their domain and purpose. They have some advantages over the simpler WSOL. However, they do not address all the issues that WSOL does. WSOL could influence the future development of WSLA

and WSML in several ways. The addition of classes of service into WSLA and WSML would make these languages more suitable for execution environments where the overhead of custom-made SLAs is too high. Explicit specification of static and dynamic relationships between classes of service and/or between custom-made SLAs would improve support for dynamic adaptation. Addition of WSOL-like reusability constructs would ease development of new WSLA or WSML files, as well as comparisons of such files. Outsourcing of definitions of QoS metrics into reusable external ontologies also supports the same goals. Addition of the formal specification of other types of constraint and statement into WSLA and WSML would make them usable for general contracts.

SLAng

Another language that can be used for specification of SLAs for Web Services is **SLAng** [Lam03]. SLAng enables specification of SLAs on the Web Service level and several lower levels, so it has broader scope than WSLA, WSML, and WSOL. However, the definitions of QoS metrics are built into SLAng schema, so SLAs have a predefined format. Adding new QoS metrics requires additions to the SLAng schema. On the contrary, definitions of QoS metrics used in WSOL are outsourced to external ontologies. In my opinion, the current version of SLAng lacks flexibility and power, so it seems less well suited for Web Services than WSLA, WSML, and WSOL. Another important difference is that development of a monitoring and management infrastructure for SLAng is in relatively early stages, while WSLA, WSML, and WSOL are already accompanied by appropriate prototype infrastructures.

3.6.2 Related Languages with the Concept of Classes of Service for Web Services

[LauT01]

At least two related works have the concept of 'classes of service' for Web Services. The first work is the presentation **[LauT01]** in which the author notes the need for a standard way to specify QoS for Web Services and suggests extending WSDL with the specification of one or more 'service classes' per WSDL port. These 'service classes' contain descriptions of QoS guarantees on the service level and on the operation level, potentially with statistical information (e.g., "in 95% of cases"). [LauT01] was published in late April of 2001, approximately at the same time as my first successful peer-reviewed paper about WSOL [Tos01]. While it identified several research topics and outlined possible solutions somewhat similar to my work, the development and implementation of the suggested solutions did not follow. Apart from some similarities between my work and the ideas presented in [LauT01], there are a number of important differences. For example, WSOL is an additional language to WSDL, while this presentation suggests extending WSDL. Further, it discusses only QoS-related topics, while the work on WSOL is broader. For these reasons, [LauT01] did not influence my work on WSOL.

WS-QoS

Web Services Quality of Service (WS-QoS) [Tia03] is another work that contains the concept of a class of service for a Web Service. It is influenced by my work on WSOL and references my papers. It attempts to address some aspects of the specification, selection, and monitoring of QoS for Web Services, particularly mapping to and from QoS of the underlying network infrastructure. The authors define WS-QoS XML schema that enables specification of several built-in QoS parameters for Web Services, several

built-in network-level QoS parameters, information about used protocols and third parties, and prices. Additional QoS parameters can be defined using WS-QoS Ontology by providing name, human-readable description, measurement unit, and whether higher or lower values are better. Further, WS-QoS can be extended with the specification of additional information. QoS is specified by both providers and consumers, so that matching of provider guarantees (offers) and consumer requirements can be performed by a Web Service Broker (WSB). WS-QoS is accompanied by a monitoring infrastructure, discussed in Section 5.9. Apart from similarities, there are considerable differences between WS-QoS and WSOL in emphasis and depth. The focus of WS-QoS is the definition of built-in QoS metrics, while WSOL outsources all such definitions to external ontologies. I agree that explicit definition of consumer QoS requirements in the same format as provider QoS guarantees is useful. While WSOL does not contain a special construct for QoS requirements, it does enable their specification. This is because QoS constraints for output-input and input-only operations are actually QoS requirements, while QoS constraints for input-output and output-only operations are QoS guarantees. Since WS-QoS specifications are relatively simple, WSOL has a number of advantages. First, WS-QoS does not provide specification of expressions and does not formally define QoS metrics. Further, it is not possible to assess from [Tia03] how detailed is WS-QoS specification of when and where QoS metrics are measured. Next, WS-QoS does not contain built-in support for periodic QoS constraints, functional constraints, access rights, monetary penalties, reusability constructs, and dynamic relationships between classes of service. WSOL specification of prices also seems more refined. For all these and other reasons,

WSOL is more powerful and more appropriate language for the specification, monitoring, and management of classes of service for Web Services.

[Marc03] and [Marc04]

The work presented in **[Marc03]** and **[Marc04]** has several similarities to my research. While it is does not formally introduce the concept of a class of service, several similar concepts are used. The authors examine e-Services as a generalization of Web Services, particularly e-Services that use multiple channels. Every channel can support several levels of QoS. Their example of an e-Service is video on demand and its channels differ in telecommunication technologies used. I think that their channels are generalization of WSDL ports, while their service levels are a limited case of classes of service. The authors discuss different aspects of high-level modeling of such e-services. Their work is still in early stages, so there are few details and no discussion of a prototype implementation. Among other topics, [Marc04] discusses quality parameters, quality sets, and quality rules. Conceptually, their quality parameters resemble WSOL QoS constraints, quality sets resemble WSOL constraint groups, and quality rules resemble WSOL static relationships between constraint groups. However, WSOL concepts are broader, specified in much more detail, and more powerful. [Marc03] discusses Service Level Specifications (SLSes) that can describe multiple service levels for multiple channels. Using an SLS as a starting point, a consumer and a provider can negotiate an SLA that will be used. Consequently, an SLS corresponds to a WSOL file with multiple service offerings. Further, [Marc03] proposes multi-channel adaptation strategies for user-triggered channel switch and provider-triggered channel switch, which lead to SLS updates. These are actually special cases of my dynamic adaptation mechanisms discussed in Chapter 4. To con-

clude, [Marc03] and [Marc04] advocate the need for exploring several research topics that I addressed with this Ph.D. research. While their discussion is high-level, my solutions are detailed.

3.6.3 Related Languages Intended for Discovery and Selection of Web Services

DAML-S

The **DAML-S (DAML-Services)** [DAM03] community works on semantic descriptions of Web Services, including specification of some functional and some QoS constraints. One of the similarities between WSOL and DAML-S is that both languages unify specification of various categories of constraints, primarily functional and QoS, and of additional information, primarily prices. There are three major differences between the two languages. First, DAML-S descriptions are intended for selection of Web Services, not for the actual monitoring and management activities. The goals of WSOL were the opposite. Second, constraints in DAML-S are currently not specified in a precise, formal, and detailed notation needed for monitoring and management activities. Third, DAML-S enables a service to provide multiple service profiles, each describing functionality and various constraints. However, DAML-S does not explicitly define the concept of classes of service that relate to the same functionality nor various static and dynamic relationships between classes of service. Consequently, I find that WSOL has a clear advantage for the management of Web Services and Web Service compositions, while DAML-S might have some advantages for discovery and selection of Web Services.

UDDIe

Several research projects suggested extending UDDI with various information to provide description and discovery of QoS for Web Services. Probably the closest to WSOL is the work on **UDDIe** [Shai03]. UDDIe extends UDDI with support for specification of a bag of user-defined properties and search based on these properties. These properties can describe, for example, QoS characteristics, price, or hardware requirements used for Grid services. Web services can be queried with simple expressions that can contain Boolean operators and value comparisons. Consequently, QoS attributes expressed as UDDIe properties can be used for Web Service discovery and selection. UDDIe also supports defining finite and infinite leases for Web Services. UDDIe is used in the G-QoSM framework for Grid computing [AlA03]. In this framework, UDDIe discovers Web Services that satisfy query criteria and a separate QoS broker selects one of them taking into consideration level of importance that a consumer assigns to particular QoS attributes. While UDDIe enables some simple specification of QoS and prices for Web Services, it has very limited capabilities. The QoS descriptions that it supports are not detailed enough for management applications. For example, information when and where to evaluate QoS constraints is missing. The decision to specify properties in WSDL files implies modification of these files whenever QoS or prices change during run-time. In addition, UDDIe does not support differentiation of service and QoS and does not have reusability constructs. Consequently, WSOL is much more powerful and flexible.

QRL

Another XML language for QoS description of Web Services intended for Web Service discovery and selection is **QRL (Quality Requirements Language)** [Mart03]. This

powerful and general language can be used not only for Web Services, but also for other distributed systems (e.g., video on demand). This generality implies complexity of the language and raises questions about its compatibility with WSDL. The main drawback of QRL is that it does not contain information about where, when, and how to perform monitoring and management of specified QoS. Further, QRL does not provide specification of classes of service and static and dynamic relationships between them.

[Mart03]

[Mart03] suggests a framework for classifying and comparing platforms for discovery and selection of Web Services. While the set of compared platforms does not include WSOL and the major related works (WSLA, WSML, SLAng, WS-QoS, DAML-S, etc.), it includes QRL, UDDIe, and some works that are not closely related to my Ph.D. research. The authors compare these platforms according to their expressiveness, formality, and viability. They assess expressiveness through support for complex clauses, support for uncertainty clauses, symmetry of specification of guarantees (offers) and requirements (demands), bilaterality of QoS specifications (in the sense that a provider can impose requirements on its consumers), and support for definition of assessment criteria for comparison of QoS values. Further, they judge formality using formal description of semantics and possibility of checking various properties (such as consistency, conformance, and optimality), as well as through relative ease of use. Finally, they evaluate viability according to the availability of a prototype, support for reusability, and run-time efficiency. The conclusion of this comparison is that their work on QRL satisfies all criteria except run-time efficiency, while the other works satisfy fewer criteria. Although the discovery and selection of Web Services is not the primary application area for WSOL, it is

interesting to see how WSOL satisfies the criteria identified in [Mart03]. WSOL provides built-in specification of complex clauses, while uncertainty can be specified using ranges and/or using statistical values (e.g., averages and variances) expressed through appropriate operation calls or special QoS metrics. While WSOL does not have the concept of a consumer demand, my language supports symmetry and bilaterality of QoS specifications because QoS constraints can be used for specification of both requirements and guarantees and because it enables specification of functional pre- and post-conditions and access rights. I believe that the assessment criteria for particular QoS metrics should not be specified in WSOL files, but in their ontological definitions [Tos02b]. Further, WSOL provides formal descriptions of guarantees and requirements and enables consistency and conformance checks. It is relatively easy to use, is accompanied by the prototype WSOL parser and the prototype of the WSOI infrastructure, contains a number of reusability constructs, and is relatively run-time efficient. Consequently, WSOL satisfies all formality and viability requirements and almost all expressiveness requirements identified in [Mart03]. It can be easily extended (e.g., with the concept of a service demand) to an excellent platform for service discovery and selection.

3.6.4 Other Related Languages

Several other recent works also recognize the importance of the formal specification of various constraints (particularly QoS) and contracts for Web Services and special types of Web Services, such as Grid Services and Semantic Web enabled Web Services. In this section, I overview some of these related works. At the end, I discuss potential WSOL influences on future standards in this area.

WS-Policy [Hon03] is a general framework for the specification of policies for Web Services. WS-Policy has a number of good features. For example, it is flexible and extensible – policies can be specified both inside and outside WSDL files. This is an advantage over WSOL. Further, it has some reusability mechanisms, such as inclusion and grouping of policies. Addition of other reusability constructs present in WSOL would be beneficial [Tos03a]. Nevertheless, it must be noted that WS-Policy is only a general framework, while the details of the specification of particular categories of policies will be defined in specialized languages. The only such specialized language currently developed is WS-SecurityPolicy. WS-PolicyAssertions can be used for the formal specification of functional constraints, but the contained expressions can be specified in any language. It is not clear whether and when some specialized languages for the specification of QoS policies, prices/penalties, and other management information will be developed. This is a serious limitation. Some unification and standardization of common elements, such as expressions, of various WS-Policy languages would reduce the overhead of supporting this framework. WSOL has such unification of expressions. WS-Policy also has no concept of a contract, SLA, or class of service. Consequently, I advocate extending WS-Policy with a specification of service offering and their static and dynamic relationships. Further, WS-Policy does not detail where, when, and how are policies monitored and evaluated. Since many policies have to be monitored and controlled during run-time, WS-Policy needs better support for management applications, including explicit specification of such management information.

[Gou03] explores simple modeling of QoS and price for Web Services. The authors praise my work on WSOL, but work in a direction more similar to [LauT01]. They model

only four attributes they identify as most important: service response time, probability of failure, single transaction price, and flat subscription price. Then, the authors suggest to formally specify these attributes in UDDI entries or WSDL files. They also discuss using SOAP intermediaries and combining QoS monitoring and brokering into one entity. This work is much narrower than my work on WSOL and still in early stages. [Gou03] targeted simplicity and it is its only advantage over WSOL. In my opinion, the expressiveness and flexibility of WSOL are more suitable for the majority of cases in which QoS for Web Services has to be specified.

[McG03] suggests extending BPEL4WS business process definitions with definitions of business performance measures, such as cost, price, cycle time, throughput, and rework. The actual values for performance measures are specified in BPEL4WS extensibility elements. These performance measures can be defined for all Web Services or only for a particular customer or customer group. The format is simple and without details. In particular, QoS metrics are not defined formally and there is no explicit information about used measurement units. Further, there is no information when, where, and how to perform measurements.

CBabel [Cerq03] is a software component Architecture Description Language (ADL) that is a part of the Reflective-Reconfigurable Interconnectable Objects (R-RIO) framework. This language adapts and extends some concepts from QML [Fro98] to describe functional components and their interaction topology, contacts with extra-functional (e.g., QoS) properties, and planned reconfigurations. While this is a language for software components and not for Web Services, it has several striking similarities with WSOL. Most importantly, a contract in CBabel can contain a number of ‘services’, which are ac-

tually similar to WSOL service offerings. A CBabel ‘service’ description specifies a functional ‘link’ and a ‘profile’ that describes QoS for this ‘link’. Further, a ‘negotiation’ clause in a CBabel contract defines relationships between ‘services’, i.e., which ‘service’ to use if the current ‘service’ cannot be attained. This is analogous to WSOL service offerings dynamic relationships (SODRs). The presence of conceptual similarities between WSOL and CBabel is one of the proofs that WSOL solutions are also applicable to general software components. While there are these conceptual similarities, WSOL is a significantly more detailed and powerful language. QoS statements in CBabel ‘profiles’ contain only high-level descriptions, such as what transport protocol is used, and not detailed and precise QoS constraints. Consequently, they cannot be used for detailed QoS monitoring and evaluation. CBabel also does not describe function constraints, access rights, prices and penalties, management responsibilities, and other constraints and statements. Further, the relationships in the CBabel ‘negotiation’ clause do not contain detailed conditions for switching of CBabel ‘services’. Consequently, they are not as precise as WSOL service offerings dynamic relationships.

To be able to perform management across a range of heterogeneous systems, it is necessary to use some de jure or de facto **standards**. I advocate development of such standards for comprehensive description and differentiation of service and QoS of Web Services, as well as Web Service Management and Web Service Composition Management activities. While I did not have resources to push WSOL through a standardization process, WSOL can influence the development of future standards in this area in several ways. In my opinion, such standards can be developed by integrating good features from WSOL, WSLA, WSML, and WS-Policy into a unified and extensible framework and

platform. Other related works might have some qualities to influence the standardization process, but to a lesser degree. I already discussed how WSOL could influence individual development of the most relevant related works. WSOL could contribute to a future integrated standard any of its benefits, advantages, and unique characteristics summarized in the next section, but I particularly emphasize service offerings, service offerings dynamic relationships, reusability constructs, and support for different types of constraint and statement in one language. These are the major WSOL contributions to the body of knowledge.

3.6.5 Benefits, Advantages, and Original Contributions of WSOL

I will now summarize the main benefits, advantages over the related work, and original contributions of WSOL. They can be classified into three groups: **expressive capabilities**, **features with relatively low run-time overhead**, and **support for management applications** [Tos03b]. They are direct consequences of the goals and requirements placed on WSOL, as discussed in Section 1.5 and Section 3.2. Several related works (e.g., WSLA and WSML) have some of these characteristics, but not all of them. Consequently, the study of these WSOL characteristics is useful for the future development of Web Service technologies.

The major features that demonstrate unique **expressive capabilities of WSOL** are:

1. WSOL enables specification of multiple classes of service for a Web Service (i.e., service offerings).

2. WSOL enables specification of different static and dynamic relationships between service offerings. This is one of the major advantages of WSOL compared to the related work.

3. WSOL enables formal specification of various types of constraint and management statement. Currently supported types of constraint belong to different categories: functional constraints, QoS constraints (including periodic QoS constraints), access rights. Currently supported types of management statement also belong to different categories: prices, monetary penalties, and management responsibilities.

4. WSOL contains the unique concept of a future-condition to describe conditions to be evaluated some time after the end of operation execution.

5. WSOL can be extended with support for additional types of constraint and management statement using XML Schema mechanisms (through the generic <*constraint*> and <*statement*> elements), without any change to the current WSOL grammar.

6. WSOL has a number of diverse reusability constructs that enable easier specification of new service offerings from similar existing ones and easier comparisons between service offerings. These reusability constructs can be categorized into the definition of a service offering, reusability elements, and reusability attributes.

7. WSOL uses reusable and extensible external ontologies with definitions of QoS metrics, measurement units, and monetary units instead of defining them inside WSOL files.

The major WSOL features with **relatively low run-time overhead** are:

1. The central concept and basic construct in WSOL is a class of service for a Web Service (i.e., service offering), instead of more powerful, but more demanding alternatives, such as an SLA.

2. WSOL enables formal specification of various types of constraint and management statement, and multiple classes of service for one Web Service in one language, instead of several separate languages. The overhead of this approach is less than the overhead when different languages are used for various categories of constraints.

3. WSOL defines the grammar of expressions used for constraints. This is a more light-weight solution than supporting one or more additional expression languages. It also prevents potential compatibility problems.

4. WSOL supports delegation of metering of QoS metrics and evaluation of constraints to specialized independent third parties acting as SOAP intermediaries or probes.

5. WSOL supports periodic evaluation of constraints, which can reduce run-time overhead and is more appropriate for some QoS metrics (e.g., availability).

6. To reduce run-time overhead, WSOL also enables unique specification of occasional evaluation of QoS constraints.

The major WSOL features that **support management applications** are:

1. Different types of constraint and management statement in WSOL support different management activities, as discussed in Section 3.4. Consequently, WSOL-enabled Web Services can be monitored, metered, controlled, accounted, and billed.

2. WSOL constraints are specified formally and precisely, in a format that can be used for automatic generation of code performing monitoring of QoS metrics and evaluation of constraints.

3. WSOL has several constructs to support specification of different management third parties.

4. WSOL has explicit support for accounting parties, due to their special characteristics among management parties.

5. WSOL contains specification of monetary penalties and service offerings dynamic relationships (SODRs) to describe what happens or could happen if some constraints are not met. The specification of service offerings dynamic relationships is unique feature of WSOL.

6. WSOL service offerings dynamic relationships (SODRs), the '*autoManipulation*' attribute of the <*serviceOffering*> element, and (to some extent) reusability constructs are important for the developed algorithms and protocols for manipulation of service offerings, explained in Chapter 4. This allows dynamically adapting a composition of WSOL-enabled Web Services without breaking the composition.

Other benefits of WSOL are:

1. WSOL is compatible with WSDL 1.1, so existing software support for provisioning of WSDL-enabled Web Services can be extended for provisioning of WSOL-enabled Web Services.

2. WSOL is complementary to WSDL 1.1, so the use of WSOL does not require abandonment or modification of WSDL files. In this way, WSOL protects and enriches past investments into WSDL.

4 Dynamic Adaptation Mechanisms using Manipulation of Service Offerings

In the preceding chapters I explained why classes of service for Web Services are useful and presented WSOL as the language for their formal specification. In this chapter, I discuss the benefits of using dynamic and autonomous manipulation of WSOL service offerings as a tool for the management of Web Services and their compositions.

I start the chapter with an overview of possible approaches to dynamic adaptation of Web Service compositions and determine the need for dynamic manipulation of service offerings. Then, in Section 4.2, I propose several dynamic adaptation mechanisms, as well as algorithms and protocols for their implementation. Operations that are used in these protocols and for other monitoring and management activities related to WSOL service offerings are grouped into Service Offerings Management (SOM) port types, summarized in Section 4.3. Analytical and experimental comparisons of proposed mechanisms with their alternatives are presented in Section 4.4 and Section 4.5, respectively. In the last section of this chapter, I discuss benefits and limitations of the proposed mechanisms and how they can be integrated with their alternatives into a comprehensive WSCM system.

4.1 Possible Approaches to Dynamic Adaptation of Web Service Compositions

Often, there is a need to dynamically and autonomously adapt a Web Service composition. For example, the adaptability of the relationships between decision support Web Services, financial analysis Web Services, and stock notification Web Services might be a very valuable feature in a turbulent stock market, as I discussed in Subsection 1.4.2.

Several approaches to dynamic adaptation of Web Service compositions are possible. The most important of these approaches, and thus the main alternatives to the dynamic manipulation of service offerings, are re-composition of Web Services, switching between Web Services, and re-negotiation of SLAs (Service Level Agreements).

Re-composition of Web Services

The most general approach to dynamic adaptation is **re-composition of Web Services**. In this approach, some entity (e.g., Web Service Composition Management software or human administrator) manages a Web Service composition. Often, an explicit description of a Web Service composition in an appropriate language, such as BPEL4WS, exists. When this management entity determines that one or more Web Services in the composition are no longer appropriate (e.g., do not perform well enough), it breaks down the Web Service composition and creates a new composition using one or more replacement Web Services. In some cases, the management entity already knows what the appropriate replacement Web Services are, but in the most general case these replacement Web Services have to be found dynamically. In the e-business Web Service composition example introduced Subsection 1.4.1, if a management entity determines that a financial analysis Web Service is no longer appropriate for decision support Web Services, it performs the re-composition. Since re-composition has been also researched for compositions of software components or modules, e.g., in the work on architecture-based software adaptation [Ore98, Pry99], it can be considered the dominant approach to dynamic adaptation.

Switching between Web Services

Switching between (provider) Web Services is a special case of the re-composition of Web Services in which a consumer acts as the management entity. In other words, the consumer determines that its provider Web Service is no longer appropriate, searches for its replacement, and chooses the new provider. In this way, an explicit description of a Web Service composition is not necessary. Unless some adapters are used, the replacement Web Service should implement the same WSDL port types as the previously used provider. For example, if a decision support Web Service determines that QoS of its provider financial analysis Web Service is no longer satisfactory, it switches to another financial analysis Web Services.

Re-negotiation of SLAs

When a consumer and a provider Web Service can negotiate a new custom-made SLA (or a similar technical contract), dynamic adaptation can be achieved with the **re-negotiation of SLAs**. Re-negotiation might involve exchange of numerous SOAP messages between the consumer and the provider. For example, if a decision support Web Service determines that QoS of its provider financial analysis Web Service is no longer satisfactory, it initiates negotiation of a new custom-made SLA by sending the provider its requirements. The provider creates a new SLA and sends it to the consumer. The consumer examines this new SLA and accepts it or submits a counter-offer. The provider might submit a counter-counter-offer and the process goes on until the parties agree on a mutually satisfactory new SLA or one of the parties decides that negotiation is futile.

Discussion of these approaches

While these three approaches are powerful and general-purpose, they are not appropriate in some circumstances. For example, re-composition of Web Services and switching between Web Services break an existing relationship between consumers and providers. This might not be appropriate for wider business reasons, e.g., when Web Service vendors are strategic partners. It might also not be appropriate when establishment of trust relationships is both complex and important, as might be the case in the financial sector. On the other hand, generation and evaluation of custom-made SLAs in the re-negotiation of SLAs might require very complex code. All three approaches require some time for dynamic adaptation. In the former two approaches, discovery and selection of replacement Web Services are the slowest steps. For the re-negotiation of SLAs, the slowest steps are multiple generation and exchange of offers and counter-offers.

4.2 Dynamic Adaptation Mechanisms Using Manipulation of Service Offerings

To achieve simple, fast, and lightweight dynamic and autonomous adaptation of a Web Service composition without breaking the existing relationship between a consumer and a provider, I propose dynamic adaptation mechanisms based on manipulation of WSOL service offerings. These mechanisms dynamically adapt which service offerings a provider supports and/or a consumer uses. They can be used between operation invocations inside one session.

Table 4.1 Notation for Management Parties in Protocols for Manipulation of Service Offerings

Notation	Meaning
C	Consumer, can also perform some monitoring of service offerings
APi	Independent accounting party used for service offering *i*, can also perform some monitoring of service offerings
MPi	Independent management parties used for service offering *i* (Note: For simplicity, I show all independent management third parties for one service offering as one block; if there is more than 1 management third party in one service offering all of them receive the same messages and send appropriate responses)
P	Provider, can also perform some monitoring of service offerings; in cases when there are two providers, *P1* is the old provider used before switching and *P2* is the new provider used after switching
SO1	Old service offering (the service offering used before switching)
SO2	New service offering (the service offering used after switching)

The mechanisms that I studied in detail and will present in this Ph.D. dissertation are:

- switching between service offerings (consumer-initiated or provider-initiated),
- deactivation of a service offering,
- reactivation of a service offering,
- deletion of a service offering, and
- creation of a new service offering.

In addition, other dynamic adaptation mechanisms based on the manipulation of WSOL service offerings and the corresponding service offerings dynamic relationships (SODRs) can be studied [Tos03c]. I looked at: allowing a consumer or a class of consumer to use a service offering, disallowing use a service offering, deactivation of a SODR, reactivation of a SODR, deletion of a SODR, and creation of a SODR.

For the implementation of the proposed dynamic adaptation mechanisms, I developed appropriate algorithms and protocols. The **algorithms** for the manipulation of service of-

ferings decide whether, what, how, and when the manipulation of service offerings should be performed. On the other hand, the message-exchange **protocols** for the manipulation of service offerings govern the coordination of management parties to achieve the manipulation. Such protocols are necessary because multiple parties (provider, consumer, and management third parties) can participate in the manipulation of service offerings. Table 4.1 summarizes the notation I use for explaining the protocols for manipulation of service offerings.

Note that my work on these dynamic adaptation mechanisms, algorithms, and protocols is closely related to WSOL and WSOI. As mentioned in Section 3.4, the crucial WSOL language support for all these mechanisms is the explicit specification of static and dynamic relationships, particularly SODRs. WSOI modules that implement the algorithms and protocols for these mechanisms will be presented in Section 5.5.

4.2.1 Initial Selection of a Service Offering

Before I present the proposed mechanisms in more depth, let me first discuss **initial selection of a service offering** – the process in which a consumer that already knows about the appropriate provider Web Service chooses one of the provider's service offerings. While this is not a dynamic adaptation mechanism in the narrow sense of the term, it is a part of the configuration of Web Service compositions. Its explanation at this time might be useful for understanding of the following subsections. Remember that in my research every consumer is a Web Service, not a human end user.

Since WSOL service offerings are predefined by the provider and not custom-made for a particular consumer, the information on how to monitor them can be deployed to the

provider and/or management third parties before interactions with consumers start. A consumer can find out about available service offerings for a particular provider in several ways. For example, if WSOL files are stored in an extended UDDI registry, the consumer can query it. Alternatively, the consumer can consult some specialized helper Web Service (e.g., a broker). The third option is that the consumer directly asks the provider. I chose to work only on the latter option, so I will hereafter focus on it.

At the very beginning of interactions with the provider, the consumer has to open a session by invoking the special operation '*openSession()*' provided by the provider. This operation creates a new session and returns the session identifier (ID) to the consumer. When the consumer invokes the '*listActiveSOsForMe()*' operation, the provider returns a WSOL file containing descriptions of all active service offerings that this consumer is allowed to use. To compare service offerings, the consumer can invoke several operations, such as '*isExtension()*'. These operations can be implemented by the consumer, the provider, or a specialized helper Web Service. When the consumer decides which service offering is most appropriate, it initializes its data structures and provides the name of the chosen service offering in an invocation of the provider's '*startWithSO()*' operation. The provider then checks that this service offering exists, is active, and the consumer is allowed to use it. If yes, it initializes its data structures and invokes the '*assignSO()*' operation of the accounting party (if the provider is not acting as the accounting party). Upon receipt of this request, the account party invokes the '*assignSO()*' operation of all management third parties for this service offering to inform them that the service offering is assigned to the session, so they can initialize their data structures. When all management

third parties are ready, the accounting party replies to the provider. Then, the provider informs the consumer that it can start using the chosen service offering.

In many circumstances, this general protocol can be shortened. For example, when the consumer already knows about available service offerings, it can simply put the name of the chosen service offering as an input parameter to '*openSession()*'.

Note that the negotiation between the consumer and the provider is very simple – the provider suggests several existing service offerings and the consumer chooses one of them. While this is less powerful than negotiation of custom-made SLAs, it requires much simpler program code. In addition, selection of service offerings can be done much faster than negotiation of custom-made SLAs, primarily because of the smaller number of exchanged SOAP messages. While this approach has limitations, it also has advantages because of its simplicity, speed, and low overhead.

4.2.2 Consumer-initiated Switching between Service Offerings

Dynamic switching between service offerings is the basic adaptation mechanism in my work. **Switching between service offerings** means changing which service offering a consumer uses. Either a consumer or a provider can initiate it, so a difference can be made between consumer-initiated switching and provider-initiated switching. WSOL service offerings dynamic relationships are used to choose the replacement service offering, particularly in the case of provider-initiated switching. In this subsection, I present consumer-initiated switching, while in the next one I discuss provider-initiated switching.

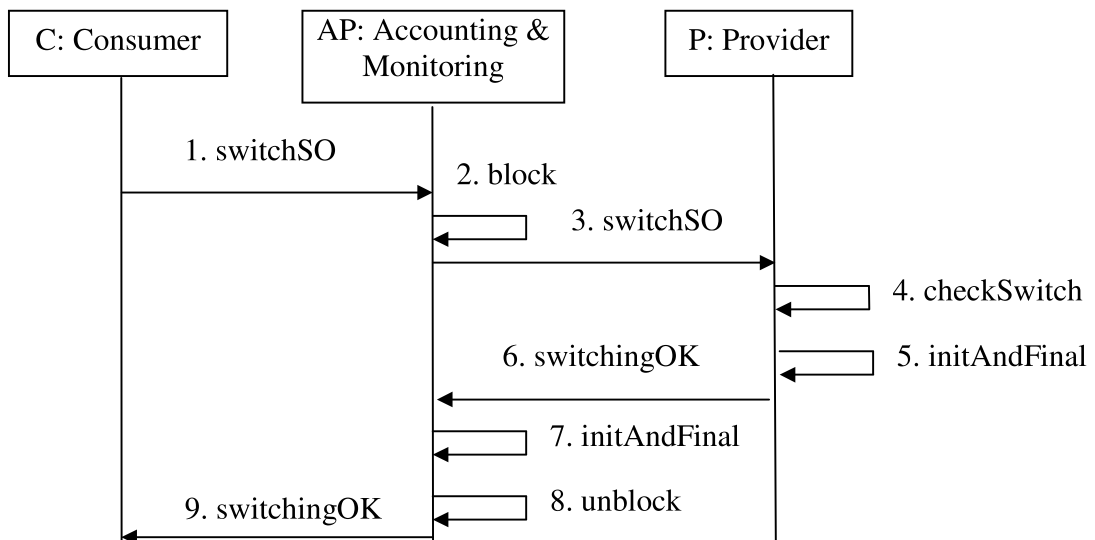


Figure 4.1 A UML Sequence Diagram with an Example Scenario of Consumer-initiated Switching between Service Offerings

A consumer can initiate switching to dynamically adapt the service and/or QoS it receives without searching for another provider Web Service. For example, depending on the analysis of the current situation in a stock market, the financial analysis Web Service could want to dynamically switch between different service offerings of the stock notification Web Service it consumes.

Figure 4.1 is a UML (Unified Modeling Language) sequence diagram that shows an example scenario of consumer-initiated switching between service offerings. In this example, all management, monitoring, and accounting activities in both the old and the new service offering are performed by one third party, denoted as *AP*. The provider Web Service *P* has at least two service offerings *SO1* and *SO2*. At the beginning of the observation, the consumer *C* uses *SO1*. However, *C* also knows about other active service offerings of *P* that it can use, particularly *SO2*. At some time, *C* decides that it would be better if it used *SO2* instead of *SO1*. Therefore, *C* sends the accounting party *AP* the '*switchSO*' message containing the name of *SO2* (Step 1 in Figure 4.1). After the receipt of this mes-

sage, *AP* blocks and queues further requests from *C* to *P* in the given session (Step 2). However, the requests that were received by *AP* before the switching request are processed using *SO1*. When *AP* finishes processing of all these requests, it sends *P* the '*switchSO*' message containing the name of *SO2* (Step 3). *P* checks whether the switching is possible, e.g., whether *SO2* exists, is active, and *C* has the right to use it (Step 4). Let me assume that the switching is possible, so *P* performs initialization of its internal activities and data structures related to *SO2* and finalization of activities and data structures related to *SO1* (Step 5). Then, *P* sends *AP* the '*switchingOK*' message as the reply to Step 3, with results of the finalization activities for *SO1* (Step 6). After it receives this message, *AP* performs initialization and finalization of its activities and data structures (Step 7). If *AP* has queued any requests from *C*, it unblocks and processes them using *SO2* (Step 8). When everything is ready for *C* to use *SO2*, *AP* sends the '*switchingOK*' message, the reply to Step 1, to *C* (Step 9). Afterwards, *C* uses *SO2*.

Let me generalize the above example scenario into a description of the protocol that is used for consumer-initiated switching between service offerings. I will examine the most general case, when monitoring of a service offering is performed by a provider, a consumer, an independent accounting party, and one or more independent management third parties. The old service offering (*SO1*) and the new service offering (*SO2*) differ in management third parties (*MP1* and *MP2*, respectively), including independent accounting parties (*AP1* and *AP2*). The other initial and final conditions for this generalized protocol are the same as for the previously discussed example scenario.

The generalized protocol for consumer-initiated switching between service offerings has to include the following sub-protocols:

1. Switching beginning (initiation): The purpose of this sub-protocol is to prepare all management parties participating in the old service offering *SO1* for switching. An important part of this preparation is to finish processing of already submitted consumer requests using *SO1* and to block and queue processing of any new consumer requests until the end of switching. Provider-side checks determining whether the switching is possible are also performed in this sub-protocol. For easier reference, I denote this sub-protocol as 'B' (for 'beginning').

2. Initialization of parties for the new service offering: The purpose of this sub-protocol is to initialize all management parties participating in monitoring activities for the new service offering *SO2*. Two examples of initialization activities are: a) making an association between the current session ID and the service offering name, and b) creation and/or initialization of appropriate data structures. I denote this sub-protocol as 'I' (for 'initialization').

3. Finalization of parties for the old service offering: The purpose of this sub-protocol is to collect all information about *SO1* in the accounting party *AP1* and finalize all *SO1*-related activities in all management parties participating in monitoring activities for the old service offering *SO1*. Examples of finalization activities are calculation of total prices and penalties to be paid and deletion of *SO1*-related data structures or their parts that are no longer needed. I denote this sub-protocol as 'F' (for 'finalization').

4. Relaying (forwarding) of requests to the new accounting party: The purpose of this sub-protocol is to transfer information about all consumer requests submitted after the switching initiation. Since *AP1* blocks and queues processing of consumer requests sub-

mitted after the switching initiation and these requests are to be processed with *SO2*, they have to be transferred to *AP2*. I denote this sub-protocol as 'R' (for 'relaying').

5. Notification of the consumer and the provider about the completed switching: The purpose of this sub-protocol is to inform the consumer and the provider that the switching has been successfully completed. I denote this sub-protocol as 'N' (for 'notification').

These sub-protocols are described in detail in Appendix A. Note that the complexity presented in Appendix A is largely due to the generality of the supported cases. In many real cases, there is no need to implement all steps of this protocol. In particular, when the provider Web Service performs all management and accounting activities, switching between service offerings becomes very simple.

Figure 4.2 is a system-level UML statechart diagram of the generalized protocol for consumer-initiated switching. Since there are many steps in the generalized protocol, this figure does not show the inner states and transitions within the sub-protocols and does not depict individual protocol participants. Nevertheless, it is useful because it shows general relationships between the sub-protocols, as well as exception handling.

Modularization into sub-protocols enables their reusability for the analogous generalized protocol for provider-initiated switching. The only presented sub-protocol that is not completely reusable for provider-initiated switching is switching beginning ('B').

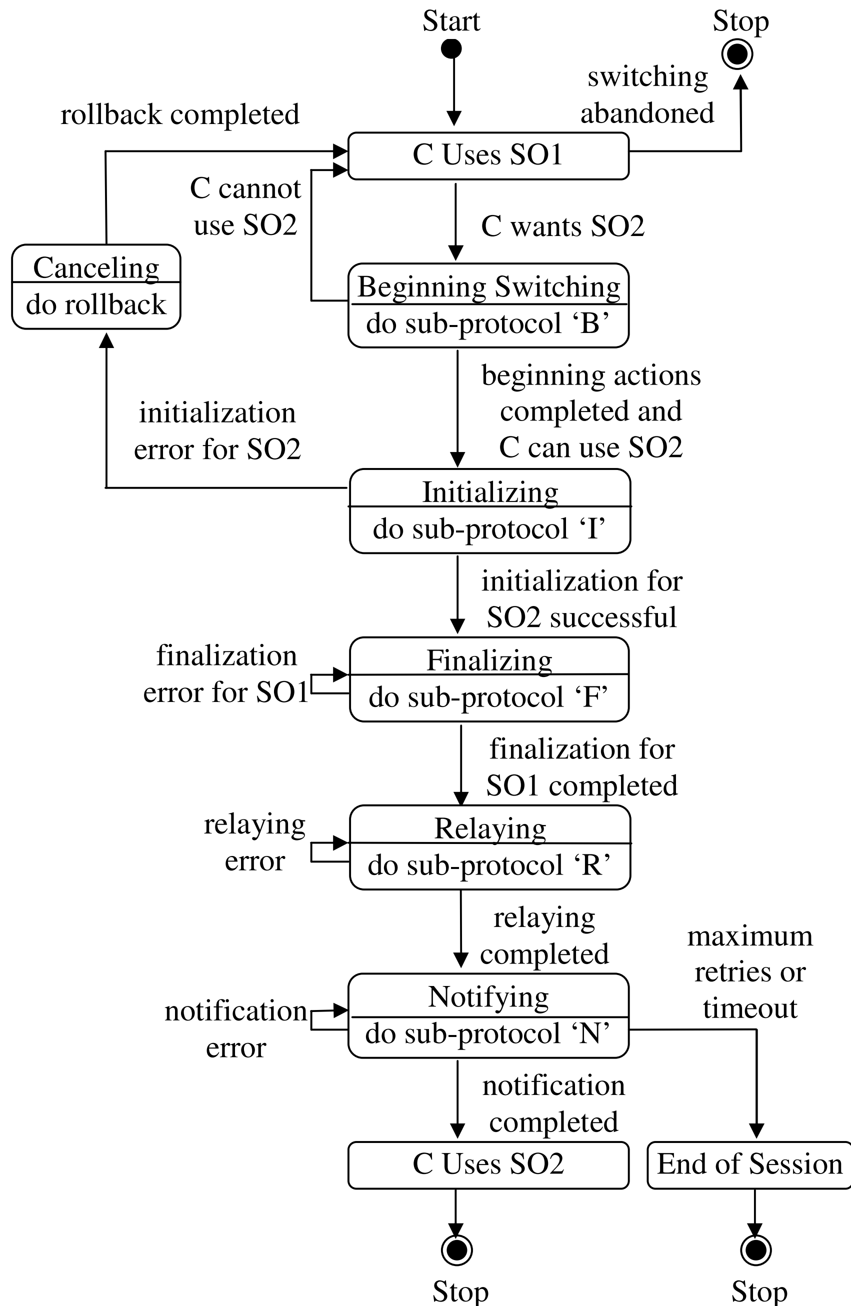


Figure 4.2 System-level UML Statechart Diagram for Consumer-initiated Switching

On the other hand, modularization introduces some redundancies. When the above modules are combined together, a number of optimizations can be applied to reduce the number of exchanged SOAP messages. For example, only one SOAP message can be

sent to request both initialization activities for the new service offering and finalization activities for the old service offering. This optimization can save a number of SOAP messages, but it also makes recovery from errors more complex. In addition, optimizations reduce reusability of sub-protocols and their implementations. Possible optimizations in this generalized protocol are also discussed in Appendix A.

Different types of exception can happen during the studied protocol. It is important to note that if an exception happens in the sub-protocols 'B' or 'I", the consumer cannot use *SO2*. An important example of such an exception is that a management party for *SO2* cannot be initialized. This means that the sub-protocols 'B' and 'I' are 'all or nothing'. If an exception happens in the sub-protocol 'I', rollback might have to be performed. The rollback procedure cancels actions at already initialized management parties for *SO2*, unblocks *AP1*, and notifies the provider and the consumer. In case of rollback, the requests queued at *AP1* will be processed using *SO1*. On the other hand, if an exception happens in sub-protocols 'F' or 'R', the consumer will be able to use *SO2* because these sub-protocols are 'best effort'. For example, if a management party for *SO1* cannot be finalized, the switching proceeds although some management information from this party might be lost. Exceptions in sub-protocol 'N' mean that the consumer and/or the provider are no longer available. Therefore, they require retries. When the maximum number of retries or timeout is reached, the session between the consumer and the provider is ended.

The described protocol for consumer-initiated switching is a representative of the protocols I developed. To save space, the other algorithms and protocols will be just summarized and not described in detail in this Ph.D. dissertation. Further details about all these algorithms and protocols are available in [Tos04e]. The most recent prototype of WSOI

contains relatively well-documented Java implementation of all these algorithms and protocols, as will be explained in Section 5.5.

4.2.3 Provider-initiated Switching between Service Offerings

A provider can initiate switching of service offerings to gracefully upgrade or degrade its service and/or QoS in case of changes. For example, after dynamic creation of a new service offering, a stock notification Web Service can suggest to some of its consumers that they switch to the new service offering. On the other hand, if QoS provided by a stock notification Web Service suddenly drops, a financial analysis Web Service that uses it might suggest to its own consumers that they switch from affected service offerings to different ones.

In the case of provider-initiated switching, the consumer is notified about the change and the replacement service offering suggested by the provider. If the '*autoManipulation*' attribute of the WSOL <*serviceOffering*> element of the old service offering is set to '*False*', the consumer is asked for confirmation of the switching before it occurs. If this '*autoManipulation*' attribute is set to '*True*', the provider only notifies the consumer after the switching. In the latter case, the consumer can reject the suggested service offering by using consumer-initiated switching to another service offering. The consumer can continue to use the old service offering, unless the provider deactivates it or disallows this consumer to use it.

To summarize, consumers can accept the suggested replacement service offering, initiate switching to another replacement service offering they prefer, or close the session with this provider. In many situations (e.g., when trust is important), it is better for an af-

fected consumer to accept the replacement service offering from the same provider than to search for another provider Web Service.

Note that the provider-initiated switching involves a business risk for the provider. If the consumer does not like the other service offerings of this provider and decides to use a competitor's Web Service instead, the vendor of the old provider Web Service loses a customer and associated revenue. Consequently, I think that abuses of this adaptation mechanism will not be frequent.

The generalized protocol for provider-initiated switching is similar to the previously described protocol for consumer-initiated switching. The main difference is in the sub-protocol 'B' for switching beginning. The protocol can be initiated in several circumstances, e.g., when unsatisfied constraints are discovered or after deactivation or reactivation of service offerings. I will now summarize one scenario. Every provider receives information from the accounting party (or an internal accounting module, if it is the accounting party) about satisfied or unsatisfied constraints. If evaluated constraints were not satisfied, the provider compares its history of unsatisfied constraints with conditions in service offerings dynamic relationships for the current service offering to determine whether any of these conditions are met. If yes, then the provider checks whether switching to the replacement service offering is possible, i.e., whether the replacement service offering is active and this consumer has the right to use it. If switching is possible, then the provider initiates the sub-protocol for switching beginning by sending the '*switchInProgress*' message to the accounting party for the old service offering, which blocks further requests from the consumer. The provider can ask for consumer confirmation of switching using the '*switchSuggested*' message. After the provider receives responses for

these messages, it starts the sub-protocols for finalization, initialization, relaying, and notification, which are identical to the ones used for consumer-initiated switching.

4.2.4 Deactivation of a Service Offering

Deactivation of a service offering means temporarily disabling its use by all consumers, including those consumers that currently use it. It is used by a provider Web service when changes in operational circumstances affect what service offerings it can provide to consumers. Some service offerings cannot be used in all situations. For example, it is sometimes impossible to achieve high QoS. An example of changed circumstances is unexpected fluctuation in the QoS provided by Web Services used by the provider. Another example is some temporary disturbance of the communication between the involved parties, e.g., due to mobility. This dynamic adaptation mechanism supports both graceful degradation and seamless service upgrades and expansions. Note that determining conditions for deactivation, reactivation, deletion, and creation of service offerings is outside the scope of my research. I only concentrated how to do these dynamic adaptation mechanisms if the need is determined.

The central issue in dynamic and autonomous deactivation of service offerings that cannot be supported in the new circumstances is what to do with consumers using the offering to be deactivated. I developed an algorithm to handle them. The essence is that the affected consumers are switched to an appropriate replacement service offering using provider-initiated switching. Further, both affected and unaffected consumers are notified about the deactivation.

The order of steps in the algorithm depends on the value of the '*autoManipulation*' attribute of the deactivated service offering. If it is '*True*', then the consumer is first switched and then notified about the switching and deactivation. In addition, if this value is '*True*' and no service offering was previously deactivated in an affected session, then the name of the deactivated service offering is stored in every such session for use in the algorithm for reactivation of service offerings. In this way, the name of the preferred service offering for this session is stored for future automatic switching during reactivation. If a service offering was previously deactivated and its replacement becomes deactivated, then the former one remains the preferred service offering for this session. The consumer can also invoke provider's '*setPreferredSO()*' operation to designate a deactivated service offering as preferred. On the other hand, if the '*autoManipulation*' attribute of the deactivated service offering is '*False*', every affected consumer is first notified about the deactivation, asked for confirmation of the switching to the replacement service offering, and switched only after the confirmation is received. In the latter case, there is no automatic storage of the name of the preferred service offering.

Note that the replacement service offering can differ between the affected consumers because a consumer might not be allowed to use every service offerings. If there is no appropriate replacement service offering to which an affected consumer can be switched, the provider might initiate creation of new service offerings. When this is also not possible, other approaches to dynamic adaptation, such as re-composition of Web Services, have to be used.

4.2.5 Reactivation of a Service Offering

Reactivation means enabling again the use of the deactivated service offering, e.g., after another change of circumstances. For example, assume that a service offering of a financial analysis Web Service was temporarily deactivated due to an unexpected reduction in the QoS provided by used stock notification Web Services. Then, when this used QoS is back into the normal range, the financial analysis Web Service can reactivate the affected service offering.

I also developed an algorithm for reactivation of service offerings. Apart from notification of all consumers about the reactivation, this algorithm includes switching to this service offering, but only for those consumers that have this service offering as preferred. The preferred service offering is determined during deactivation or explicitly designated by the consumer, as discussed in the previous sub-section. After the provider determines that the reactivated service offering is preferred for a particular session, it examines the '*autoManipulation*' attribute of the currently used replacement service offering. If this attribute is '*False*', then the consumer is notified about reactivation and asked for confirmation of switching to the reactivated service offering. If this attribute is '*True*', the consumer is first switched, then notified. Automatic switching to the reactivated service offering can help in achieving, as much as possible, the originally intended level of service and QoS.

4.2.6 Deletion of a Service Offering

If the probability of future reactivation of a deactivated service offering is zero or very low, the provider Web Service can decide to dynamically delete it. **Deletion of a service**

offering permanently removes support for it, e.g., used data structures. Only deactivated service offerings can be deleted. For example, if implementation of the stock notification Web Service is dynamically upgraded to improve performance, some of its service offerings with lower QoS might become redundant and can be deleted. Another example is when a management third party used in some service offering goes out of business.

4.2.7 Creation of a Service Offering

Dynamic creation of new service offerings can be used after a change in the implementation of the provider Web Service, in the Web Services that the provider uses, in management third parties, in the execution environment, or in consumer needs. This dynamic adaptation mechanism provides further adaptability of Web Services and Web Service compositions. While the dynamic adaptation mechanisms discussed above handle changes that are to some extent anticipated, the creation of new service offerings can be used for unanticipated changes. Not all circumstances of run-time operation, particularly QoS, and not all consumer needs can be predicted in advance. In addition, Web Services and Web Service compositions can evolve (e.g., be upgraded) dynamically. Creation of new service offerings provides some support for evolution with minimal disruption of the operation and for propagation of changes to co-operating Web Services.

For example, when the implementation of the stock notification Web Service is dynamically upgraded, then new service offerings of this Web Service can be created. To reflect these changes, new service offerings of the financial analysis Web Service can also be created. This propagates upgrade benefits from the stock notification Web Service to consumers of the financial analysis Web Service, e.g., decision support Web Services.

While these are examples of provider-initiated creation of service offerings, there is also a possibility of consumer-initiated creation in special cases. An example of such special case is a request from an important consumer that brings in substantial revenue for the provider side.

Note that creation of a new service offering is not creation of new functionality or new functional interfaces (i.e., WSDL port types), but creation of new sets of constraints and management statements for the existing functionality. However, dynamic creation of new service offerings can be non-trivial and can incur non-negligible overhead. It cannot be performed arbitrarily due to various possible conflicts. For example, QoS constraints cannot be set arbitrarily because of the limitations of used resources (including other Web Services), mutual dependencies of QoS parameters, and other issues. Detection and resolution of such conflicts can be very complex. Creating new service offerings might consume considerable time and resources of the Web Service.

Consequently, I only addressed dynamic creation of a new service offering in some simple and limited special cases. I concentrated on creation of new service offerings as variations of existing service offerings, based on WSOL reusability elements and attributes, and on leveraging the modular architecture of WSOI. WSOI uses reusable and parameterizable modules called 'WSOI-specific handlers', described in detail in Section 5.3. I explored creation of new WSOL service offerings that can be achieved with only re-use, re-parameterization, and/or reordering of these modules, because this can be supported even without a fully implemented WSOL compiler.

Let me illustrate such dynamic creation of service offerings on an example. Assume that a stock notification Web Service has two operations: '*stockValue*' returns a value for

one input stock symbol, while '*multipleStockValues*' returns array of values for an input array of stock symbols. Service offering *SO1* contains an instantiation of a constraint group template that guarantees that response time for '*stockValue*' must be less than '*20 milliseconds*', but there is no response time guarantee for '*multipleStockValues*'. A new service offering *SO2* can be created in such a way that its contents is the same as for *SO1*, but that it instantiates the mentioned constraint group template with '*15 milliseconds*' and thus strengthens the response time guarantee for '*stockValue*'. Re-parameterization of the module that evaluates this response time QoS constraint adapts WSOI to monitor *SO2*. Additionally, a new service offering *SO3* can be created as an extension of *SO1* that adds a QoS constraint that guarantees that response time for '*multipleStockValues*' will be less than '*40 milliseconds*'. Re-parameterization and reuse for '*multipleStockValues*' of the module that evaluates this QoS constraint for '*stockValue*' adapts WSOI to monitor *SO3*. In both cases, there is no need to generate and deploy new modules. Only descriptions of which modules are used for a particular context and in what order have to be updated.

The cases and complexities related to dynamic creation of service offerings that I did not address can be a topic for interesting future research.

4.2.8 Other Possible Mechanisms for Dynamic Manipulation of Service Offerings

An important group of additional mechanisms is **deactivation, reactivation, deletion, and creation of service offerings dynamic relationships (SODRs)**. I informally studied these mechanisms, primarily in the context of manipulation of service offerings. After a service offering is deactivated, reactivated, or deleted, the corresponding SODRs have to be deactivated, reactivated, or deleted. When a new service offering is created, new

SODRs can be created, while some old ones might be deactivated or even deleted. However, the relationship between the manipulation of SODRs and the manipulation of involved service offerings can be more complex. For example, assume the stock notification Web Service provides three service offerings and at least one SODR. The SODR states that if service offering *SO3* is deactivated because the response time guarantee for the operation '*stockValue()*' cannot be kept, then the appropriate replacement service offering is *SO2*. After several deactivations and subsequent reactivations of *SO1* due to the response time guarantee for '*stockValue()*', the provider Web Services (or some entity managing it) notices that the vast majority of consumers prefer service offering *SO1* as a replacement for *SO3*. Then, a new SODR can be created and the old one can be deactivated. Another pair of dynamic adaptation mechanisms related to service offerings is **allowing and disallowing particular consumers or classes of consumer to use some service offerings**. I left such security-related mechanisms for future research.

4.3 Service Offering Management (SOM) Port Types

To achieve monitoring of WSOL service offerings and particularly their manipulation, it is necessary to coordinate the involved management parties. In particular, the protocols that govern the coordination of management parties for manipulation of service offerings contain a number of special operations to be implemented by participating parties and exposed to other parties. For example, the provider-side operation '*switchSO()*' shown in Figure 4.1 is invoked by the accounting party in consumer-initiated switching. It is also useful to enable that selected external management software or human administrators can invoke the suggested mechanisms for manipulation of service offerings. Further, opera-

tions such as '*listActiveSOsForMe()*' mentioned in Subsection 4.2.1 are significant for selection of WSOL service offerings. While the majority of monitored WSOL-related management information is exchanged using SOAP headers, special operations for explicit exchange of this information are needed. For example, after a management party measures a periodic QoS metric and/or evaluates a periodic QoS constraint, it can use such an operation implemented by another management party to inform it about the results. Management parties can implement another operation that can be used to inform them about the values of metered or calculated QoS metrics, evaluated WSOL constraints, and their monetary consequences.

For all these reasons, I defined the signatures of a number of operations that participate in the protocols for manipulation of service offerings, as well as other externally-accessible operations related to selection, monitoring, and manipulation of WSOL service offerings. Then, I grouped these operations into several WSDL port types, which I named **Service Offerings Management (SOM) port types** [Tos03c, Tos04a].

Table 4.2 gives a brief explanation of all SOM port types and shows what management parties should implement them. *SOInfo* and *SOComparisons* are used for selection of WSOL service offerings. *SessionMgmt*, *SONotification*, *SOM_Prov*, *SOM_MgmtP*, and *SOM_AccP* are used for monitoring of WSOL service offerings. *SOM_Prov*, *SOM_MgmtP*, *SOM_AccP*, *SOM_Cons*, *SODRMgmt*, and *SOSecurity* are used in the discussed protocols for dynamic adaptation.

Table 4.3 describes one operation from every SOM port type. Figure 4.3 shows the most important operations from all SOM port types. Full signature of these and other SOM operations can be found in [Tos04e].

Table 4.2 Explanation of Service Offering Management (SOM) Port Types

Port Type Name	Imple-mented by	Explanation
SessionMgmt	Providers	Operations for session management
SOInfo	Providers	Operations about available service offerings and their activity
SOComparisons	Providers	Operations for determining static relationships between service offerings; used during selection of Web Services and service offerings
SOM_Prov	Providers	Provider-specific operations for monitoring and manipulation of service offerings
SOM_AccP	Accounting parties	Operations that accounting parties use to participate in monitoring and manipulation of service offerings
SOM_Cons	Consumers	Consumer-specific operations for monitoring and manipulation of service offerings
SOM_MgmtP	All management parties	Operations that all management parties (providers, accounting parties, management third parties, and consumers) implement to participate in monitoring and manipulation of WSOL service offerings
SONotification	All management parties	Operations used to exchange (pull or push) WSOL-related management information
SODRMgmt	Providers	Provider-side operations for information about service offerings dynamic relationships and their manipulation
SOSecurity	Providers	Operations for security management related to service offerings

Several characteristics of the SOM port types should be noted. First, the definition of well-known (ideally: standardized) management operations significantly reduces the complexity of management activities. Several other works, such as WSLA [Lud03], Open Grid Services Architecture (OGSA) [Fos02], WS-Agreement [Cza03], and WS-Manageability [Pot03] also define management operations for Web Services.

Table 4.3 Explanation of Representative Operations in SOM Port Types

Port Type Name	Operation Name	Explanation
SessionMgmt	openSession	Opens a session between the provider implementing this operation and the consumer invoking it; returns the session ID (identification) number
SOInfo	listSOsForMe	Returns a WSOL file describing all WSOL service offerings that the consumer can use
SOComparisons	listExtensions	Returns names of all available WSOL service offerings that are extensions of the service offering whose name is provided as parameter in the operation invocation
SOM_Prov	startWithSO	Used by a consumer at a beginning of a session to select a service offering; assigns a service offering to the current session
SOM_AccP	forwardRequests	Used during switching of service offerings when the old and the new service offering have different accounting parties; forwards to the new accounting party all consumer requests queued at the old accounting party
SOM_Cons	switchSuggested	Used during provider-initiated switching of service offerings; invoked by the provider to suggest a replacement service offering to the consumer; the consumer can accept the suggested service offering, suggest another replacement service offering, or close the session
SOM_MgmtP	listSOMOps	Returns a list of all SOM operations that the management party implements
SONotification	inform	Used to push WSOL-related management information to the management party that implements this operation
SODRMgmt	deactivateSODR	Deactivates the service offerings dynamic relationship the name of which is supplied as the operation parameter
SOSecurity	allowSO	Gives a consumer the right to use a service offering; consumer name and service offering name are provided as parameters; invoked by Web Service management entities

SessionMgmt
openSession() closeSession()

SOInfo
listSOsForMe() descCurrentSO() listActiveSOs() listAllSOs()

SOComparisons
isExtension() listExtensions() doesInstantiate() doesInclude() compare()

SOM_Prov
startWithSO() switchSO() prInitSwitchSO() deactivateSO() reactivateSO() createSO() deleteSO() setPreferredSO()

SOM_AccP
switchSO() switchInProgress() switchingTo() forwardRequests() readHistory() readBalance()

SOM_Cons
switchInProgress() switchSuggested() switchingTo() soDeactivated() soReactivated() soCreated() soDeleted()

SOM_MgmtP
deploySO() assignSO() initializeWork() finalizeWork() initAndFinal() switchCancelled() listSOMOpsForMe()

SONotification
inform() readValue()

SODRMgmt
listSODRsForMe() listAllSODRs() listActiveSODRs() deactivateSODR() reactivateSODR() createSODR() deleteSODR()

SOSecurity
allowSO() disallowSO()

Figure 4.3 Example Operations in Service Offering Management (SOM) Port Types

Second, Web Services do not describe implementation, but only the signature of supported operations. Consequently, the implementation of management-related operations for Web Services can be provided independently from the implementation of port types describing business (functional) logic of the Web Service. In particular, this implementation can be provided by monitoring and management infrastructure. Further, some or all Web Services of the same vendor can share the actual implementation of management operations. In my work, WSOI implements SOM operations for all WSOL-enabled Web Services. Further, all Web Services using the same WSOI instance share the code of this implementation and to some extent data structures. This lowers the overhead of supporting SOM port types.

Third, modules within the same management entity (e.g., provider) can invoke SOM operations using internal programming language calls, without any SOAP message.

Fourth, since provider Web Services are essential participants in all mechanisms for dynamic manipulation of service offerings, the majority of SOM port types are to be implemented by them. Some port types are to be implemented by consumers, accounting parties, and management third parties. In addition, SOM operations from *SOInfo*, *SOComparisons*, and *SODRMgmt* port types that provide information about available service offerings and their static and dynamic relationships can be implemented by specialized Web Service brokers.

Fifth, a Web Service need not support all discussed dynamic adaptation mechanisms. For example, some provider Web Services might not implement creation and deletion of service offerings and service offerings dynamic relationships. Consequently, a management party (e.g., a provider) might implement only some SOM port types and/or only some operations within a particular port type. Nevertheless, to participate in dynamic manipulation of WSOL service offerings, management parties, particularly providers, have to implement at least some operations from SOM port types.

Sixth, management operations can be provided through separate management Web Services or through separate ports. Both approaches have advantages and disadvantages. The advantages of defining management port types are easier discovery and easier support for only some SOM port types and operations in them. A disadvantage is that when SOM operations are first added to an existing Web Service or their signature is modified, it is necessary to update the WSDL file describing this Web Service. However, such situations are rare and independent from changes to service offerings. While WSOL, WS-Manageability, OGSA, and WS-Agreement define management ports, WSLA defines separate management Web Services. The concept of management interfaces (port types)

was also advocated and explored in several earlier works on the management of distributed services, e.g., [Slo95].

Seventh, for the majority of operations from SOM port types, only selected external entities should be allowed access. These entities are primarily the other management parties in the composition, but potentially also special management software or human administrators that manage the provider and/or the whole Web Service composition. For example, consumers can initiate switching and, potentially, dynamic creation of service offerings. Further, they participate in provider-initiated switching and should be notified about the results of the other dynamic adaptation mechanisms initiated by providers. However, they should not be allowed to initiate these other mechanisms, such as deactivation of a service offering. These other mechanisms, and thus the SOM operations for them, should be invoked only by the provider or an external management entity. In addition, some provider Web Services might allow only particular consumers or classes of consumer to initiate switching of service offerings, e.g., due to performance or security reasons. Such restrictions could be permanent or limited to special circumstances.

4.4 Analytical Comparisons of the Proposed Mechanisms and Their Alternatives

As already mentioned, the main alternatives to the mechanisms proposed in this chapter are re-composition of Web Services, switching between (provider) Web Services, and re-negotiation of Service Level Agreements (SLAs). Manipulation of service offerings deals with a limited number of service offerings and involves communication only with already known provider Web Service and management third parties. Consequently, it seems as a

simpler, faster, and more lightweight (in terms of less run-time overhead) approach to dynamic adaptation than re-negotiation of SLAs and re-composition of Web Services.

To formally estimate the complexity, overhead, and delay and, thus, assess the benefits and limitations of my proposed mechanisms, I analytically compared consumer-initiated and provider-initiated switching between service offerings with switching between Web Services, as well as with re-negotiation of SLAs. On the other hand, Wei Ma and I conducted experiments comparing consumer-initiated and provider-initiated switching between service offerings with switching between Web Services. These analytical and experimental comparisons are important validations of the suggested dynamic adaptation mechanisms. Other analytical and experimental comparisons can be conducted in future research. In this section, I describe my analytical comparisons, while in the next section I present the experimental comparisons. Summaries of these analytical studies, experiments, and conclusions were also published in [Tos03c] and [Tos04a].

Since no methodology for analytical estimation of complexity, overhead, and delay of various dynamic adaptation approaches for Web Services was previously reported in the literature, I had to define such a methodology. Analytical estimations in this area can consider a number of issues. However, the generation, transmission, and processing of SOAP messages incur the biggest overhead and delay in using many Web Services [Lauk03, Syc03]. Experiments that Wei Ma and I conducted confirm that internal operations and manipulation of data structures performed by parties involved in dynamic adaptation are relatively simple and fast compared to the generation, transmission, and processing of SOAP messages.

On the other hand, the delay of transmitting SOAP messages between parties certainly depends on a number of factors, one being distance between communicating nodes. For example, the delay incurred when two computers in the same lab communicate should be smaller than when two computers on different continents communicate. However, in some odd cases it might be vice versa. I have not done concrete measurements to obtain estimates about delays for SOAP messages for different distances. However, it seems to me that by adopting the assumption that all management parties belong to different businesses and are located in the same city or other geographic area, I can simplify estimations of complexity, overhead, and delay. The consequence of the stated assumption is that all management parties are on similar distances, so the delay of SOAP messages between them is relatively similar and can be treated as equal.

Therefore, to analytically estimate the complexity, overhead, and delay of the studied dynamic adaptation approaches, I abstracted all internal invocations and manipulations, as well as delays of individual SOAP messages, and only studied and compared **the number of SOAP messages** exchanged between the involved management parties.

I generated formulas that determine the number of exchanged SOAP messages for different dynamic adaptation mechanisms. To derive these formulas, I first defined generalized protocols and studied their optimizations, such as those discussed in Appendix A. I verified these formulas on a number of example scenarios by comparing numbers generated by formulas and numbers that I counted on UML sequence diagrams I drew for these scenarios. Table 4.4 explains symbols that are used in these formulas. These symbols represent the numbers of various management parties, as well as whether the old and new service offering use the same management third parties.

Table 4.4 Notation Used in Analytic Comparisons of the Proposed Dynamic Manipulation Mechanisms and Their Alternatives

Notation	Meaning
P_1, P_2, …, P_i	This number can be 0 or 1. If the provider (supplier) performs some measurements and/or calculations of QoS metrics and/or evaluation of constraints in service offering *i* and the provider is not the accounting party, this number is 1. In other cases, it is 0.
C_1, C_2, …, C_i	This number can be 0 or 1. If the consumer performs some measurements and/or calculations of QoS metrics and/or evaluation of constraints and/or is the accounting party in service offering *i*, this number is 1. In other cases, it is 0.
A_1, A_2, …, A_i	The number of independent accounting parties (i.e., accounting parties outside the supplier/provider and the consumer) in service offering 1, service offering 2, etc. This can be either 0 (if the accounting party is inside the supplier/provider or consumer) or 1 (it there is an independent accounting party).
$D_{1:2}$, $D_{1:3}$, …, $D_{i:j}$	This number can be 0 or 1. If the accounting party in service offering 1 is different from the accounting party in service offering 2, then $D_{1:2}$ is 1, otherwise it is 0. For $D_{1:3}$ accounting parties in service offerings 1 and 3 are compared, etc.
$S_{1:2}$, $S_{1:3}$, …, $S_{i:j}$	$S_{1:2}$ is the number of independent accounting parties that are used in both service offering 1 and service offering 2. This number can be calculated as: $\min(\min(A_1, A_2),(1-D_{1:2}))$. Effectively, it is between 0 and 1. For $S_{1:3}$ independent accounting parties in service offerings 1 and 3 are compared, etc.
M_1, M_2, …, M_i	The number of independent management parties (i.e., third-parties outside the supplier/provider, the consumer, and the independent accounting party) in service offering 1, service offering 2, etc. This number can be 0, 1, 2, …
$E_{1:2}$, $E_{1:3}$, …, $E_{i:j}$	$E_{1:2}$ is the number of independent management parties that are used in both service offering 1 and service offering 2. This number can be betewen 0 and $\min(M_1, M_2)$. For $E_{1:3}$ independent management parties in service offerings 1 and 3 are compared, etc.
O	Number of messages exchanged between the consumer and the provider to negotiate the contents of a new custom-made SLA. Since I assume request-reply communication and since at least one SLA offer has to be made, this number must be an even number greater or equal 2.

Table 4.5 Formulas for Switching between Service Offerings and Switching between Web Services

Protocol	Number of Exchanged SOAP Messages
Consumer-initiated Switching of service Offerings, Modularized (CSOM)	$2*(2+3*A_1+1*M_1+1*C_1+1*A_2+1*M_2+0*C_2+1*D_{1:2})$
Consumer-initiated Switching of Web services, Modularized (CSWM)	$2*(3+3*A_1+1*M_1+1*C_1+1*A_2+1*M_2+0*C_2+1*D_{1:2})$
Difference CSOM – CSWM	-2*(1)
Consumer-initiated Switching of service Offerings, Not modularized (CSON)	$2*(1+2*A_1+1*M_1+1*C_1+1*A_2+1*M_2+0*C_2+1*D_{1:2})$
Consumer-initiated Switching of Web services, Not modularized (CSWN)	$2*(2+2*A_1+1*M_1+1*C_1+1*A_2+1*M_2+0*C_2+1*D_{1:2})$
Difference CSON – CSWN	-2*(1)
Provider-initiated Switching of service Offerings, Modularized, with Confirmation (PSOMC)	$2*(2+3*A_1+1*M_1+1*C_1+1*A_2+1*M_2+0*C_2+1*D_{1:2})$
Provider-initiated Switching of service Offerings, Modularized, No confirmation (PSOMN)	$2*(1+3*A_1+1*M_1+1*C_1+1*A_2+1*M_2+1*C_2+1*D_{1:2})$
Provider-initiated Switching of Web services, Modularized, No confirmation (PSWMN)	$2*(2+3*A_1+1*M_1+1*C_1+1*A_2+1*M_2+0*C_2+1*D_{1:2})$
Difference PSOMN–PSWMN	$-2*(1-1*C_2)$
Provider-initiated Switching of service Offerings, Not modularized, with Confirmation, (PSONC)	$2*(2+1*A_1+1*M_1+0*C_1+1*A_2+1*M_2+0*C_2+1*D_{1:2})$
Provider-initiated Switching of service Offerings, Not modularized, No confirmation (PSONN)	$2*(1+1*A_1+1*M_1+0*C_1+1*A_2+1*M_2+0*C_2+1*D_{1:2}$ $+1*max(C_1,C_2))$
Provider-initiated Switching of Web services, Not modularized, No confirmation (PSWNN)	$2*(3+1*A_1+1*M_1+0*C_1+1*A_2+1*M_2+0*C_2+1*D_{1:2})$
Difference PSONN–PSWNN	$-2*(2-1*max(C_1,C_2))$

I first analytically compare switching between service offerings and switching between Web Services. Table 4.5 shows the most important formulas that I generated for these comparisons. The table covers consumer-initiated and provider-initiated switching of service offerings and Web Services, modularized and not modularized. Detailed explanations of how all these formulas were generated are given in [Tos04e]. As explained in Appendix A, the generalized protocols are 'modularized', while their 'not modularized' versions apply some overlapping between sub-protocols to avoid obvious redundancy. While I gave formulas for both confirmed and unconfirmed provider-initiated switching of service offerings, I viewed provider-initiated switching of Web Services analogous only to the latter. For switching between Web Services, I assumed that the consumer does not spend time in searching for the replacement provider *P2*, because it already knows about it (e.g., the old provider *P1* advises the consumer about the appropriate replacement).

From these formulas, it is clear that switching between service offerings requires fewer exchanged SOAP messages, and is thus faster, than switching between Web Services. (For modularized provider-initiated switching without confirmation switching of service offerings and switching of Web Services can have the same number of SOAP messages in some cases. However, as soon as optimizations are applied, switching of service offerings always requires fewer SOAP messages.) The savings are between $2*(1-C_2)$ and $2*(2-1*\max(C_1,C_2))$ messages, which means that they are between 0 and 4 SOAP messages, with 2 SOAP messages being the most frequent saving. The relative importance of these savings depends upon the number of management parties, particularly

management third parties. As the number of management third parties grows, the relative importance of savings decreases.

For example, when all monitoring and management activities are performed only by the provider and the protocol implementation is not modularized, $A_1=M_1=C_1=A_2=M_2=C_2=D_{1:2}=0$. Consequently, the number of exchanged SOAP messages for consumer-initiated switching of service offerings is 2, while for switching of Web Services is 4. This is a relatively big difference of 50%. However, when $A_1=C_1=A_2=C_2=D_{1:2}=1$ and $M_1= M_2=3$ and there is no modularization, the number for consumer-initiated switching between service offerings is 24, while for switching between Web Services is 26. The difference is 7.7%.

I also compared similar formulas for cases when other optimizations are used and the conclusions are similar. To achieve objectivity, I tried to apply similar optimizations, to the extent possible, to both compared sides. Some optimizations can even improve the benefits of the switching between service offerings. In particular, it is easier to combine initialization and finalization for switching between service offerings than for switching between Web Services.

These formulas also show that the delay of the proposed dynamic adaptation mechanisms linearly increases with the number of involved management parties. Further, these formulas indicate that the delay does not depend on the number of supported service offerings. However, the latter conclusion might be misleading because it is a consequence of my abstraction of the internal operations of Web Services. If the number of supported service offerings, SLAs, or similar contracts is huge (e.g., in thousands or even millions), then it is reasonable to expect some degradation of performance and increase of delays.

I also compared switching between service offerings with negotiation of a new custom-made SLA, both consumer- and provider-initiated. These protocols are similar – activities that have to be performed for service offerings also have to be performed for SLAs or at least have a direct counterpart. For example, deployment of the new SLA to all management parties requires the same number of SOAP messages as the initialization of a new service offering. However, there is an important difference that service offerings are predefined. There is no negotiation about the contents of a service offering, only selection. On the other hand, some additional messages have to be exchanged to determine the contents of a new custom-made SLAs. In a trivial case, one side (usually the provider) suggests the contents of the new SLA and the other side agrees. The number of exchanged SOAP messages in this case is 2, the same as for informing to which service offering to switch. In other scenarios, the process of determining and agreeing about the contents of the new SLA can take many more steps, including exchange of counter-offers. The more SOAP messages are exchanged in this process, the greater the savings of service offerings. To conclude, the number of exchanged SOAP messages for consumer- and provider-initiated creation of a new SLA can be calculated as:

the number for analogous switching of Web Services + (O-2).

Note also that negotiation of SLAs introduces complexity and overhead that is not captured in the above formula because I abstracted the operations performed at particular management parties. Providers and consumers have to analyze and reason about offers and counter-offers for custom-made SLAs and this is more complex than simple selection of a predefined service offering. Further, generation of custom-made SLAs, even when it is based on templates, has to take into consideration potential dependencies and conflicts

between the contained SLOs. While the number of exchanged SOAP messages for deployment of an SLA to independent management parties and for initialization activities for service offerings is the same, the former requires bigger SOAP messages and more complex activities by every involved party.

Let me also briefly compare dynamic creation of service offerings and negotiation of SLAs. Creation of a service offering can be viewed as a special case of the negotiation of a custom-made SLA (or another technical contract) and its substitute. Both processes result in a new contract and can involve matching consumer's preferences with provider's capabilities. However, there are also differences. First, a provider can create a new service offering without consumer input, while negotiation of SLAs implies some form of consumer involvement. Second, negotiation of a new SLA implies that the consumer that negotiates it will use this SLA, while the service offering is, in principle, not created for a particular consumer (although a particular consumer might initiate the creation) and need not be immediately used by any consumer. Third, WSOL service offerings are both simpler and broader than SLAs because they do not contain definitions of QoS metrics and some management information present in SLAs, but contain additional types of constraint. This may make their negotiation both simpler and more complex. However, it seems that definition and comparison of new QoS metrics is more open-ended, and thus more complex, than definition and comparison of new functional constraints and access rights. Fourth, WSOL reusability constructs enable easier creation and comparisons of WSOL service offerings. They can reduce the complexity, delay, and overhead measured through the amount of work on a particular management party and the number of exchanged SOAP messages. Many of these reusability constructs, such as extension, are not

present in the majority of SLA-oriented languages. While both creation of service offerings and negotiation of SLAs can be implemented with different levels of complexity—from almost trivial to extremely sophisticated and powerful—creation of service offerings can leverage the mentioned simplifications that are not present for SLAs. The cases of creation of service offerings described in Subsection 4.2.6 and supported in the current WSOI prototype do not perform negotiation of the service offering contents and generation of new WSOI modules, so they are examples of simplicity.

4.5 Experimental Comparisons of the Proposed Mechanisms and Their Alternatives

The conclusion of analytical comparisons that manipulation of service offerings is simpler and faster than alternatives was confirmed or complemented by experimental comparisons. In these experiments [Tos04a, Tos03c], average delay and average run-time Java Virtual Machine (JVM) memory usage were measured for switching between service offerings and switching between Web Services, both consumer- and provider-initiated, in comparable scenarios. To get a more accurate picture, for both comparisons several experiments representing different scenarios were conducted. While I designed these experiments, they were implemented and conducted by Wei Ma.

Note that it was not a goal of these experiments to additionally validate the formulas for switching between service offerings and for switching between Web Services presented in the previous section. One of the reasons is that the measured delay in these experiments is influenced by other factors, additional to the number of exchanged SOAP messages. Nevertheless, there is a relatively good correspondence between the results of

the analytical and the performed experimental comparisons. A more through study can be conducted in the future.

To conduct these experiments, Wei Ma set up a test-bed environment with several Web Service compositions in a local network. Some of these Web Services used the Web Service Offerings Infrastructure (WSOI) extension of Apache Axis, while some used only Apache Axis, both of which will be explained in Chapter 5. Both WSOI and Axis are run over Apache Tomcat application server. Wei Ma also set up a test UDDI directory for experiments in which discovery of Web Services was performed dynamically.

Let me now discuss a representative experiment that compared consumer-initiated switching between Web Services with consumer-initiated switching between service offerings. In this experiment, a financial analysis (FA) Web Service invokes (and thus consumes) a stock notification (SN) Web Service to find out current values of stock symbols.

In the first case, shown in Figure 4.4.a), there are two stock notification Web Services, *SN1* and *SN2*, implementing the same WSDL port type, but with different service offerings. *SN1* has *SO1*, while *SN2* has *SO2*. A financial analysis Web Service *FA1* uses *SN1* and its *SO1*. However, after some time, *SO1* is no longer appropriate for *FA1*. Therefore, *FA1* closes the session with *SN1* and opens a session with *SN2*. After this switching, *FA1* continues to use *SN2* and its *SO2*. In reality, *FA1* would use a UDDI directory or another Web Service discovery mechanism to discover that *SN2* provides the required WSDL port type and *SO2*. In this simple example, this information is hardcoded into *FA1*.

Figure 4.4 a) Switching of Web Services; b) Switching of Service Offerings

The previous case using simple switching between Web Services is compared with the case that uses switching between service offerings, shown in Figure 4.4.b). In this second case, *SN1* provides two service offerings, *SO1* and *SO2*, and *FA1* first uses *SN1* with *SO1*. *FA1*, *SO1*, *SO2*, and the WSDL description of *SN1* are the same as in the first case. When the need for dynamic adaptation arises, *FA1* switches from using *SO1* to using *SO2* without closing the session with *SN1*. After the switching, *FA1* uses *SN1* and its *SO2*.

Table 4.6 describes configuration of this experiment, while Table 4.7 contains the measured results. The consumer and the providers executed on the same computer. For simplicity, the information about the appropriate replacement Web Service or replacement service offering was hardcoded into consumer's implementation. Averages for delay and JVM memory usage were calculated using 100 test runs. For switching between Web Services, 4 SOAP messages had to be exchanged: '*closeSession*' and its reply, and '*openSession*' and its reply. The average delay was 28 ms. For switching between service offerings, 2 SOAP messages were enough: '*switchSO*' and its reply. The average delay in this case was 13 ms, which is about 54% less than for switching between Web Services. An analytical comparison of these two scenarios predicts the relative difference of 50%. This is a good correspondence, having in mind that in this experiment both Web Services execute on the same computer and there is no actual message exchange over the network.

Table 4.6 Description of an Experiment Comparing Consumer-initiated Switching between Web Services and Switching between WSOL Service Offerings

Description	Switching of Web Services	Switching of Service Offerings
Number of Web Services	3 (consumer and 2 providers)	2 (consumer and provider)
Distribution	Same computer	Same computer
Provider Web Services	Simple stock notification Web Service (both providers)	Simple stock notification Web Service
Infrastructure for providers	Tomcat, 2 * standard provider-side Axis modules (2 Axis engines – one per provider), 2 * *WSOISessionHandler*	Tomcat, standard provider-side Axis modules, *WSOISessionHandler*, WSOI-specific modules (WSOI version 1)
Infrastructure for consumers	Standard consumer-side Axis modules, *WSOISessionHandler*	Standard consumer-side Axis modules, *WSOISessionHandler*
How the consumer finds the replacement WS or SO	Hardcoded into consumer's implementation	Hardcoded into consumer's implementation
Number of exchanged SOAP messages	4 ('*closeSession*' and its reply, '*openSession*' and its reply)	2 ('*switchSO*' and its reply)
Start time for measuring delay	The consumer sends to the old provider '*closeSession*' message	The consumer sends to the provider the '*switchSO*' message
Stop time for measuring delay	The consumer receives from the new provider the reply for the '*openSession*' message	The consumer receives from the provider the reply for the '*switchSO*' message
Software participating in JVM memory usage	Tomcat, 2 * standard provider-side Axis modules, standard consumer-side Axis modules, 3 * *WSOISessionHandler*, implementation of the consumer and both providers	Tomcat, standard provider-side Axis modules, standard consumer-side Axis modules, 2 * *WSOISessionHandler*, other WSOI-specific modules, implementation of the consumer and the provider

Table 4.7 Results of the Experiment Comparing Consumer-initiated Switching between Web Services and Switching between WSOL Service Offerings

Measured Value [Units]	Switching between Web Services (= A)	Switching between Service Offerings (= B)	Difference (= B-A)	Relative Difference (= (B-A)/A) [%]
Delay [ms]	28	13	- 15	- 53.57 %
JVM memory usage [MB]	8.84	6.89	- 1.95	- 22.05 %

The JVM memory usage was calculated as a sum of memory used on consumer side and on provider side. For switching between Web Services, the provider infrastructure contained Tomcat, standard Axis provider-side modules, only session management modules from WSOI, and implementation code. While the two providers in this case used different Axis instances (which were both counted in JVM memory usage), they used the same Tomcat instance and the Tomcat memory overhead was counted only once. For switching between WSOL service offerings, the provider infrastructure contained Tomcat, standard Axis provider-side modules, relevant WSOI modules, and implementation of the stock notification Web Service. In both cases, consumer infrastructure contained standard Axis consumer-side modules, WSOISessionHandler, and simple implementation code. The average total JVM memory usage for switching between Web Services was about 8.84 Megabytes. For switching between WSOL service offerings, the average total JVM memory usage was about 6.89 Megabytes, which is about 22% less.

Similar results were obtained in other experiments comparing consumer-initiated switching between Web Services and switching between service offerings. For example, in an experiment with somewhat different configuration, consumer-initiated switching of service offerings was about 18% faster and consumed about 4% less memory.

Table 4.8 contains description of an experiment comparing provider-initiated switching between Web Services and provider-initiated switching between WSOL service offerings. Table 4.9 contains the measured results for the described experiment. The consumer and provider executed on the same computer. For switching of service offerings, the provider searched WSOI data structures to find to which replacement service offering to switch the consumer. For switching of Web Services, the old provider searched simple

internal data structures to recommend a replacement provider Web Service, so that the consumer had no need to perform search for a replacement Web Service.

In this experiment, provider-initiated switching between service offerings was about 15% faster and consumed about 17% less memory. Note that an analytical comparison predicts that switching between service offerings would be 50% faster in this case. The difference can be explained with the fact that in the experiment both Web Services execute on the same computer and there is no actual message exchange over the network, while my analytical comparisons assume exchange over a network.

Similar results were obtained in other experiments comparing consumer-initiated switching between Web Services and switching between service offerings. For example, in an experiment with somewhat different configuration, provider-initiated switching of service offerings was about 13% faster and consumed about 21% less memory.

The two described experiments deliberately use only very simple selection of the replacement Web Service. This makes switching between Web Services relatively simple and straightforward. A number of additional complexities are present in reality. Two examples of possible additional complexities are when the consumer and the provider execute on separate computers and/or when switching between Web Services requires searching a UDDI directory to find the replacement Web Service. When such additional complexities are introduced, the advantages of the switching between service offerings are greater. This observation was confirmed by our additional experiments [MaW04].

Table 4.8 Description of an Experiment Comparing Provider-initiated Switching between Web Services and Switching between WSOL Service Offerings

Description	Switching of Web Services	Switching of Service Offerings
Number of Web Services	3 (consumer and 2 providers)	2 (consumer and provider)
Distribution	Same computer	Same computer
Provider Web Services	Simple stock notification Web Service (both providers)	Simple stock notification Web Service
Infrastructure for providers	Tomcat, 2 * standard provider-side Axis modules (2 Axis engines – one per provider), 2 * WSOISessionHandler	Tomcat, standard provider-side Axis modules, WSOISession-Handler, WSOI-specific modules (WSOI version 1)
Infrastructure for consumers	Standard consumer-side Axis modules,WSOISessionHandler	Standard consumer-side Axis modules,WSOISessionHandler
Number of exchanged SOAP messages	4 ('*sessionClosed*' and reply, '*openSession*' and reply)	2 ('*switchSuggested*' and reply)
How the consumer finds the replacement WS or SO	The old provider recommends replacement	The provider recommends replacement
How the provider finds the replacement WS or SO	Searches simple internal data structures	Searches WSOI data structures
Start time for measuring delay	The old provider determines during accounting that some constraint was not satisfied	The provider determines during accounting that some constraint was not satisfied
Stop time for measuring delay	The consumer receives from the new provider the reply for the openSession message	The provider finishes switching the consumer to the new service offering
Software participating in JVM memory usage	Tomcat, 2 * standard provider-side Axis modules, standard consumer-side Axis modules, 3 * WSOISessionHandler, implementation of the consumer and both providers	Tomcat, standard provider-side Axis modules, standard consumer-side Axis modules, 2 * WSOISessionHandler, 1 * other WSOI-specific modules, implementation of the consumer and the provider

Table 4.9 Results of the Experiment Comparing Provider-initiated Switching between Web Services and Switching between WSOL Service Offerings

Measured Value [Units]	Switching between Web Services (= A)	Switching between Service Offerings (= B)	Difference (= B-A)	Relative Difference (= (B-A)/A) [%]
Delay [ms]	319	271	- 48	- 15.05 %
JVM memory usage [MB]	8.98	7.41	- 1.57	- 17.48 %

It would be useful to also conduct experiments comparing switching between service offerings and re-negotiation of SLAs, both consumer-initiated and provider-initiated. While I had plans to include results of such comparisons into this Ph.D. dissertation, several issues warrant postponement of these experiments for future work. The crucial issue is how to conduct fair and representative experiments. If the existing management infrastructures using SLAs for Web Services are used for comparisons with WSOI using service offerings, the differences in measured delay and memory overhead may be consequences not only of the compared dynamic adaptation mechanisms, but also of the implementation of used tools. Since the general architecture and implementation maturity are significantly different between WSOI and the existing infrastructures that could be used in such comparisons, extreme caution is needed. In addition, learning the intricacies of languages and tools used for these comparisons would require considerable time. On the other hand, implementing a WSOI-based management infrastructure for custom-made SLAs also requires considerable time. Since I had to spend my available research time on tasks with higher priority, these comparisons had to be left for future work.

4.6 Discussion of Benefits and Limitations of the Proposed Mechanisms and Their Potential Integration with the Alternatives

The conducted analytical studies and experiments support the conclusion that manipulation of service offerings is generally **simpler, faster, and incurs less run-time overhead** (e.g., memory overhead) than the re-composition of Web Services and the re-negotiation of SLAs. Further, it provides additional flexibility and enhances robustness of the relationship between a provider Web Service and its consumer. This robustness is important, for example, when the consumer trusts the current provider, but does not yet trust alternative providers. In many situations, application of the proposed mechanisms can be more practical than use of their alternatives. One notable example is when providers execute in mobile systems, because the introduced delay and overhead of transmitting SOAP messages in such environments are especially high.

However, compared to the re-composition of Web Services, manipulation of service offerings has **limitations**. Service offerings of one Web Service differ only in constraints and management statements, which might not be enough for adaptation. Further, appropriate alternative service offerings cannot always be found or created. Therefore, I advocate manipulation of service offerings as a complement to, and not a complete replacement for, the re-composition of Web Services. On the contrary, it can be either a complement to or a lightweight replacement for the re-negotiation of SLAs, because they have relatively similar limitations.

Consequently, I suggest that a management system for dynamic adaptation of Web Service compositions integrates the manipulation of service offerings and the re-composition of Web Services. When a need for dynamic adaptation arises, such system

would first try to find a replacement service offering from the same provider Web Service. Only when this is not possible, the system would try to find a replacement provider Web Service and perform re-composition. In some cases (e.g., to achieve uninterrupted service), the old provider Web Service can supply a temporary replacement service offering while the consumer searches for another, more appropriate, Web Service. Such a comprehensive system for management of Web Service compositions could also implement other approaches to dynamic adaptation, such as re-negotiation of SLAs, to address cases when these approaches are more appropriate than manipulation of service offerings and re-composition of Web Services. A possible classification of different approaches to dynamic adaptation of distributed component compositions, including Web Service compositions, and ways to integrate them were discussed in [Tos02a].

5 Web Service Offerings Infrastructure (WSOI)

To enable and demonstrate monitoring of WSOL-enabled Web Services and dynamic manipulation of WSOL service offerings, I designed the corresponding management infrastructure – the **Web Service Offerings Infrastructure (WSOI)** [Tos04a, Tos03c]. WSOI monitoring activities include measurement and calculation of used QoS metrics, evaluation of WSOL constraints, and accounting of executed operations and evaluated constraints. On the other hand, WSOI implements the algorithms and protocols for dynamic manipulation of service offerings discussed in the previous chapter.

In this chapter, I first discuss the requirements for my work on WSOI. Then, in Section 5.2, I give a high-level overview of how WSOI is used in a provider Web Service and classify its modules. In the following section, WSOI modules implementing monitoring of service offerings are elaborated and illustrated with examples. Monitoring of service offerings using management third parties is discussed in Section 5.4. In the following section, WSOI-specific data structures and other modules used for dynamic manipulation of service offerings are explained. I end the chapter with a comparison of WSOI and the most important related works.

5.1 The Requirements I Placed on WSOI

Functional requirements for WSOI

Most of the requirements for WSOI are direct consequences of the general goals for this Ph.D. research, the choices about the domain of my work, and the secondary goals for WSOI stated in Section 1.5. For example, one of the consequences of the secondary goal that WSOI should allow external involvement in the manipulation of WSOL service of-

ferings is that it should provide appropriate externally available operations, e.g., implement SOM port types. Several other requirements can be concluded from the stated research goals and choices. I also determined several additional functional and non-functional requirements for the infrastructure. These functional requirements for WSOI were:

1. To **support management third parties**, primarily those that act as SOAP intermediaries. The support for management third parties acting as probes was classified as important, but not a priority due to the issues associated with using probes in business-to-business environments, to be discussed in Section 5.4.

2. To **support sessions** in such a way that the actual implementation of business-related WSDL operations of provider Web Services hosted by WSOI need not support sessions. The use of sessions for service offerings was explained in Section 2.3.

Non-functional requirements for WSOI

The additional non-functional requirements for WSOI were:

1. To have a **modular design** in which modules specific to a particular service offering can be **automatically generated** by a WSOL compiler and **easily integrated** into the overall infrastructure. These modules measure and/or calculate particular QoS metrics, evaluate WSOL constraints, and perform accounting.

2. To **reuse and extend appropriate existing tools** for Web Services, ideally open-source and supporting the sessions and SOAP intermediaries. Such reuse enables building WSOI upon proven solutions and concentrating on original characteristics (‘competitive advantages’) of WSOL and WSOI.

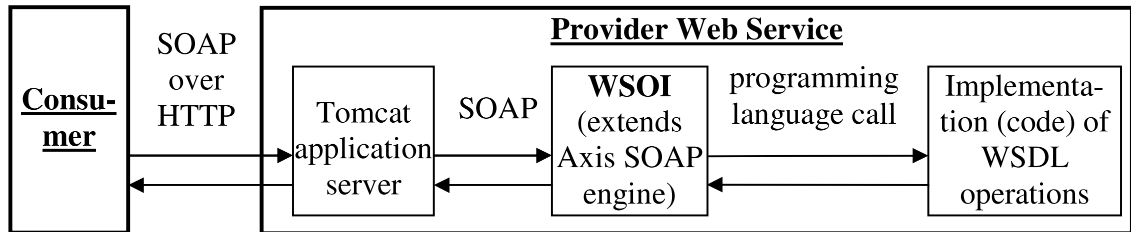


Figure 5.1 Position of WSOI inside a Provider Web Service

5.2 Overview of WSOI

The part of WSOI that performs monitoring of WSOL service offerings is based on extensions of Apache Axis (Apache eXtensible Interaction System) version 1.0 [Axi02, Tec02, Web02], a popular open-source SOAP engine implemented in Java. A SOAP engine is an application that receives, processes, and sends SOAP messages. In other words, Axis enables provisioning (i.e., hosting) of Web Services and SOAP communications with these Web Services. Axis contains the standard functionality of hosting Web Services described in WSDL and processing of their SOAP messages. WSOI contains standard Axis modules and adds modules specific to the provisioning, monitoring, and dynamic manipulation of WSOL service offerings. It is common and convenient to run Axis using the popular Apache Tomcat open-source application server.

Axis has several good features that I found crucial for WSOI [Tos03c]. Here I emphasize the three most important. First, Axis has a modular, flexible, and extensible architecture based on configurable chains of pluggable SOAP message processing components, called handlers. Such an architecture enables implementation of the monitoring of WSOL service offerings using a set of additional handlers and handler chains easily plugged into Axis. Second, Axis defines a SOAP message processing node that can be used for provider Web Services, consumer Web Services, and SOAP intermediaries such as WSOL

management third parties. Consequently, it can be used as a basis for all WSOL management parties. Third, Axis has built-in support for sessions that can be extended for WSOI needs.

Figure 5.1 shows how WSOI is used in a provider Web Service. Consumer requests are transported over a network and received on the provider side by Tomcat, which passes the SOAP content to WSOI. WSOI processes the SOAP message, performs monitoring activities, and invokes appropriate Java code that implements the WSDL operations of the provider Web Service. The results are returned to WSOI, which processes them and performs additional monitoring activities. Afterwards, Tomcat sends the reply to the consumer. A scenario of using WSOL and WSOI with management third parties will be presented in Section 5.6.

Note that one Axis instance, and thus also a WSOI instance, can be used for parallel provisioning of more than one provider Web Service. Additionally, since it includes Axis, WSOI can also be used for provider Web Services that do not support WSOL.

Figure 5.2 shows the modules in WSOI. WSOI modules can be classified into WSOL service offering monitoring modules (in the left half of the figure), WSOL service offering manipulation modules (in the right half), and modules used for both purposes (crosscutting both halves) [Tos03c, Tos04a]. In Section 5.3 I will discuss WSOI modules implementing monitoring of WSOL service offerings. In Section 5.5 I will discuss WSOI modules implementing algorithms and protocols for dynamic manipulation of WSOL service offerings, as well as WSOI-specific data structures used for both monitoring and management activities.

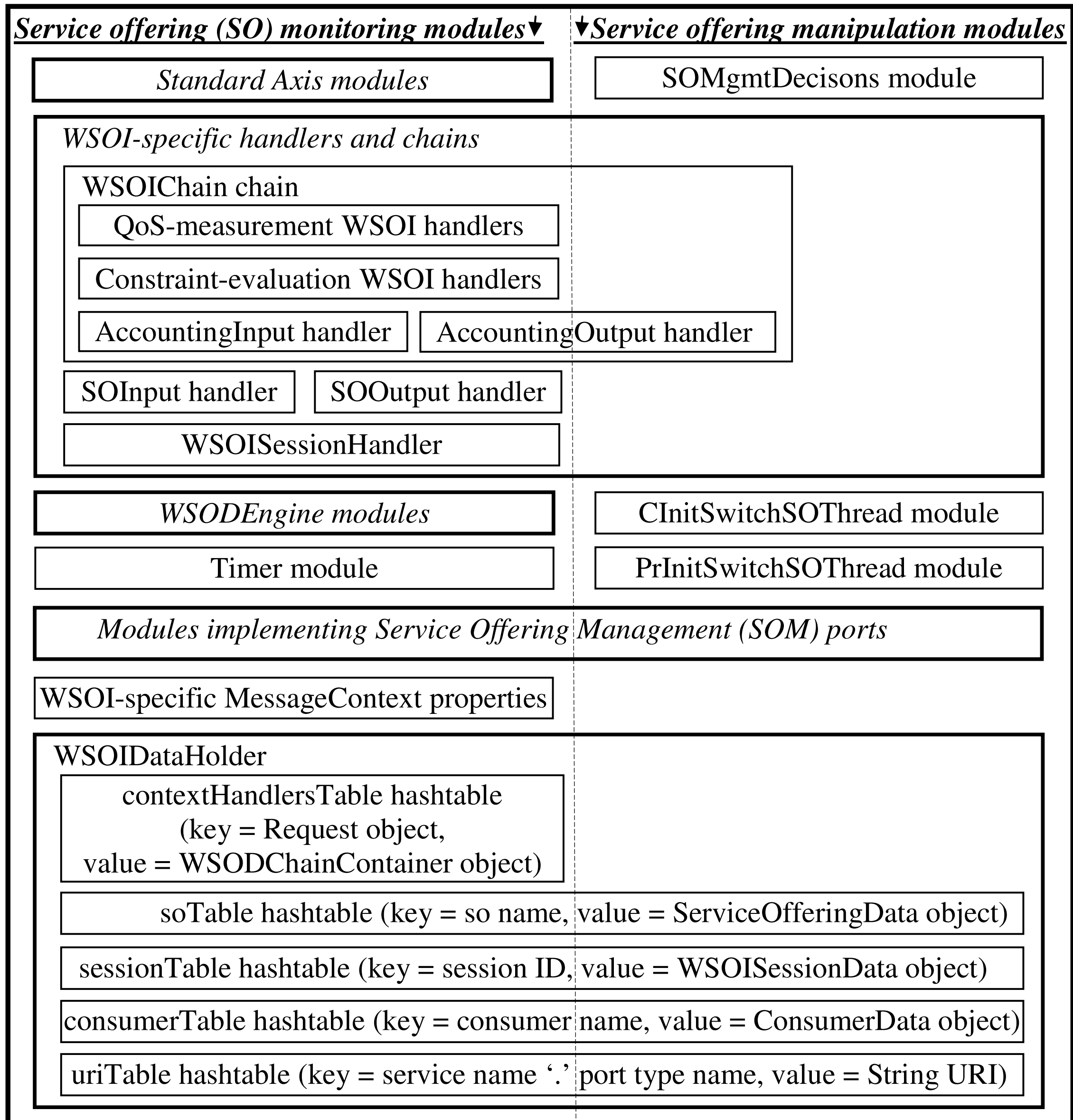


Figure 5.2 The Most Important Modules in the Web Service Offerings Infrastructure (WSOI)

There are two major versions of the proof-of-concept prototype implementation of WSOI. Wei Ma implemented WSOI version 1 to address the most important aspects of monitoring and manipulation of service offerings, while I implemented additional support

for manipulation of service offerings in WSOI version 2. Unless noted otherwise, in this Ph.D. dissertation I present the latter version of the WSOI prototype.

5.3 WSOI Modules Implementing Monitoring of WSOL Service Offerings

Since WSOI extends Apache Axis, I first summarize the Axis modules that are reused or extended by WSOI modules for monitoring of service offerings. An Axis **handler** can process input, output, and/or fault SOAP messages. For example, it can perform measurement of QoS metrics or evaluation of constraints. A handler can also alter the processed SOAP message, e.g., add/remove headers. An Axis **chain** is an ordered, pipelined collection of handlers. Since the *Chain* class is a subclass of the *Handler* class, every Axis chain is also a handler. A transport chain performs processing related to the transport of SOAP messages. A global chain performs other processing applicable to all Web Services. A service chain performs processing characteristic for a particular Web Service. Axis handlers exchange information through an instance of the ***MessageContext*** class, which contains information about the request message, the response message, and a set of properties. Axis handlers can use the *MessageContext* properties for decisions related to SOAP message processing and can modify these properties.

WSOI adds specialized Axis handlers performing WSOL-related measurement and calculation of QoS metrics, evaluation of constraints, calculation of prices and penalties to be paid, and accounting activities. Hereafter, I refer to these handlers and their chains as '**WSOI-specific handlers and chains**'. One example is the *AccountingOutput* (*AO*) handler that performs calculation of prices and penalties, accounting activities, and initiation of provider-initiated switching. While this WSOI-specific handler is used for both

monitoring and dynamic manipulation of service offerings, the other WSOI-specific handlers are used only for monitoring activities. Almost all WSOI-specific handlers and chains are subclasses of the abstract *WSOIHandler* class, which extends the standard Axis class *BasicHandler* with several data members and methods useful for all WSOI-specific handlers. The only exception is the *WSOISessionHandler* class, which extends the standard Axis class *SimpleSessionHandler*.

There is a correspondence between WSOL constructs and the management activities performed in WSOI-specific handlers, as discussed in Section 3.4. I designed the WSOI-specific handlers so that a WSOL compiler can generate them automatically from WSOL files. However, since the prototype WSOL compiler was not yet fully implemented, Wei Ma manually implemented some of these handlers for the WSOI prototype.

I studied several designs for WSOI-specific handlers that evaluate WSOL constraints. One possible approach is for a WSOL compiler to create generic expression-evaluation handlers that can be used for many different WSOL constraints. However, to reduce run-time overhead, I chose that a WSOL compiler should generate a specialized handler for every WSOL constraint, except for parameterized constraints specified inside WSOL constraint group templates. For every parameterized constraint that is specified inside a WSOL constraint group template, a WSOL compiler generates a parameterized WSOI-specific handler class. Instantiation of a constraint group template with some concrete parameter values results in constructing and using a new instance of this class. The *WSOIHandler* class defines data members for storage of these parameter values and methods for their easier use in subclasses.

There are also several possible designs for WSOI-specific handlers that measure or calculate QoS metrics. I chose that a WSOL compiler should generate these handlers from ontological definitions of QoS metrics. As mentioned in Section 3.5, I left the research of using existing distributed system management technologies to collect values for measured QoS metrics for future work.

In different invocation contexts, different WSOI-specific handlers are used. An **invocation context** is determined by the full name of the invoked operation (containing Web Service, port, and operation names), the name of the service offering, and the name of the management party in which this handler is used, and whether the processed message is request, response, or fault. In WSOL files, constraints and management statements are associated with invoked operations and service offerings using applicability domain attributes, containment of constraints and management statements inside service offerings. Further, they are associated with management parties using management responsibility statements and the '*accountingParty*' attribute of service offerings. Additionally, the type of the constraint or management statement determines whether it is relevant for the input and/or the output message flow. In this way, WSOL files contain information necessary for determining the invocation context of a constraint or a management statement.

Wei Ma developed a special XML file format, called the **Web Service Offering Descriptor (WSOD)**, for describing the order of WSOI-specific handlers used in a particular context. As discussed in Section 3.5, WSOD files can be generated by a WSOL compiler, at the same time as when WSOI-specific handlers are generated. While a WSOL compiler might use a limited number of precedence rules for determining the order of the handlers (e.g., that preconditions are evaluated before QoS constraints for request mes-

sages), it seems that there is a possibility of error. Therefore, human Web Service administrators can also generate or modify WSOD files. Inside WSOI, the information about the used WSOI-specific handlers and their order is stored in the ***contextHandlersTable* hashtable**. The key for this hashtable is a description of the context, while the value is an object that contains all WSOI-specific handlers used for processing the request and response message. The ***WSODEngine* module** within WSOI loads this ordering information from WSOD files into *contextHandlersTable*. Further details are available in [MaW04].

When a new service offering is created, the corresponding classes for WSOI-specific handlers and WSOD descriptions produced by a WSOL compiler can be deployed to management parties with the '*deploy()*' operation from the *SOM_MgmtP* SOM port type. For simplicity, this distribution is performed manually in the current WSOI prototype.

The ***WSOIChain*** class is the crucial WSOI module for the monitoring of WSOL-enabled Web Services. It contains code that examines what is the invocation context and dynamically constructs the chain of appropriate WSOI-specific handlers for the given context based on the contents of the *contextHandlersTable* hashtable. In this way, different WSOI-specific handlers are used for different WSOL service offerings and/or different operations. An instance of the *WSOIChain* class is a WSOI-specific sub-chain of the Axis' service chain. An alternative approach to using *WSOIChain* would be to introduce the fourth type of chain (in addition to transport, global, and service chains) into Axis – a 'service offering chain'. The solution based on the *WSOIChain* was adopted due to its relative simplicity.

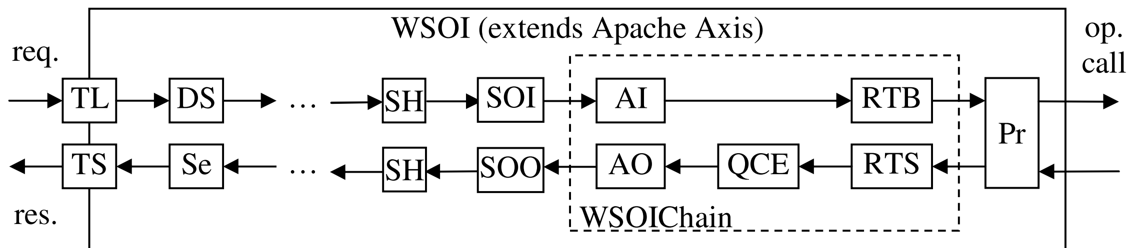
Several WSOI-specific handlers are used by all management parties. For efficiency reasons, they are implemented outside the *WSOIChain* chain. One of them is *WSOISessionHandler* (*SH*) that performs WSOL-related session management activities. For processing input messages, *WSOISessionHandler* reads session ID (identifier) from a SOAP message header and writes it into a *MessageContext* property. It also checks that this session ID is valid and determines the service offering used for this session. For processing output messages, *WSOISessionHandler* reads the session information from this *MessageContext* property and writes it into a SOAP header. Note that in WSOI, sessions are opened and closed using operations from the *SessionMgmt* SOM port type that are built into WSOI and provided independently from the hosted Web Services. The actual implementation of business-related WSDL operations of the hosted Web Service need not additionally support sessions, although it may.

Further, the WSOI-specific handlers *ServiceOfferingInputHandler (SOI)* and *ServiceOfferingOutputHandler (SOO)* translate the management information between *MessageContext* properties and SOAP headers. While special *MessageContext* properties are used to transport WSOL-related management information between WSOI modules (particularly, WSOI-specific handlers), special SOAP message headers are used to transport this information between different management parties acting as SOAP intermediaries. The transported management information can include results of measurement or calculation of QoS metrics, evaluation of constraints, and accounting of prices/penalties to be paid. *WSOIDeserializer* is a part of *ServiceOfferingInputHandler* that reads information from SOAP message headers and writes it into special properties of the current *MessageContext* instance. *WSOISerializer* is a part of *ServiceOfferingOutputHandler* that

writes WSOL-related management information from *MessageContext* properties into appropriate SOAP message headers. Both *WSOIDeserializer* and *WSOISerializer* were designed and implemented by Wei Ma.

Figure 5.3, shows an example configuration of handlers inside the provider-side WSOI used for monitoring of WSOL-enabled Web Services. In this example, the provider Web Service measures response time, evaluates a QoS constraint limiting the measured response time, and performs accounting. The arrows in the figure represent passing of information and control between internal modules, achieved through local procedure calls. At the beginning of processing of an input (request) message, the standard Axis module *TransportListener* (*TL*) receives the request SOAP message and passes it to the standard Axis handler *Deserializer* (*DS*). *Deserializer* creates a *MessageContext* instance (hereafter simply referred to as 'message context') used by other handlers.

After *Deserializer*, several other standard Axis handlers used for all Web Services are executed [Web02]. Due to space limits and the fact that these handlers in transport and global chains are not crucial for the illustration of WSOI, they are not shown in Figure 5.3. Instead, their position is indicated with '...'. Next, for every Web Service that supports WSOL, the WSOI-specific handlers *WSOISessionHandler* (*SH*), *ServiceOfferingInput* (*SOI*), and the chain *WSOIChain* are executed in the Axis service chain. In this example, the first handler in *WSOIChain* for the input message is *AccountingInput* (*AI*), which records the request message. Then, the *ResponseTimeBegin* (*RTB*) handler stores into message context the start time for measuring response time. After this, the standard Axis handler *Provider* (*Pr*) is executed outside *WSOIChain*. It dispatches the call to the Java object implementing the requested operation of the Web Service.



Legend of Symbols

req.	request (input) message	res.	response (output) message
TL	TransportListener	TS	TransportSender
DS	Deserializer	Se	Serializer
…	Standard Axis modules, executed for all Web Services	SH	WSOISessionHandler
SOI	SOInput	QCE	QoSConstraintEvaluation
AI	AccountingInput	AO	AccountingOutput
WSOI-Chain	WSOIChain	SOO	SOOutput
RTB	RequestTimeBegin	RTS	RequestTimeStop
Pr	Provider	op. call	Java call to the implementation of the Web Service's operation

Figure 5.3 An Example Configuration of Handlers and Message Processing inside Provider-side WSOI

This Java object returns its results back to the *Provider* handler. After *Provider*, handlers in the *WSOIChain* process the output (i.e., response) message. First, the handler *ResponseTimeStop* (*RTS*) stores into message context the stop time for measuring response time, as well as the difference between this stop time and the start time stored by *ResponseTimeBegin*. Next, the QoS constraint limiting response time is evaluated in the handler *QoSConstraintEvaluation* (*QCE*). This handler stores its results into the message context. Finally, the *AccountingOutput* (*AO*) handler uses the information from the message context to calculate prices and penalties to be paid. This information is also stored into the message context. After *WSOIChain*, *ServiceOfferingOutputHandler* (*SOO*) and *WSOISessionHandler* are executed. Next, several standard Axis handlers used for all Web Ser-

vices, not shown in Figure 5.3, are executed in Axis global and transport chains. The last of these handlers is *Serializer* (*Se*), which packs information from the message context into SOAP. The *TransportSender* (*TS*) module of Axis sends the response SOAP message to the consumer. The management information about the measured response time, the evaluated QoS constraint, and prices or penalties to be paid is in SOAP headers.

When for another service offering and/or for another operation different QoS metrics have to be measured, different WSOL constraints evaluated, and/or different prices or penalties calculated, then only the contents of the *WSOIChain* changes appropriately. The configuration of all other handlers remains the same.

The measurement or calculation of periodic QoS metrics and evaluation of periodic constraints differs from the example in Figure 5.3. It is initiated by the accounting party. More precisely, it is initiated by ***Timer***, a special active module in WSOI for accounting parties. If the provider acts as the accounting party (as in the discussed example), *Timer* is a module in provider-side WSOI. It keeps a list of periodic activities and when some of the times stored in the list occurs, it invokes a *WSOIChain* instance, which creates and executes a chain of appropriate WSOI-specific handlers. The results of such a measurement, calculation, and/or evaluation can be stored locally for future processing. They can also be reported to other management parties in a special notification SOAP message. Since active modules that perform periodic invocation of other programs are already well-researched, e.g., in operating systems, the implementation of the designed *Timer* module was not a priority for the current WSOI prototype. Similarly, WSOI support for occasional measurement of QoS metrics and evaluation of QoS constraints, described

with the WSOL '*evalPeriod*' attribute, was designed, but not implemented in the current WSOI prototype. These modules will be a part of a future WSOI implementation.

The described monitoring of WSOL-enabled Web Services with WSOI incurs some run-time overhead. I designed and Wei Ma implemented and conducted a number of **experiments** to demonstrate feasibility and usefulness of using WSOL and WSOI for monitoring of Web Services and to estimate the overhead that such monitoring places on Web Services. The scenarios in these experiments are related to the '*buyStock*' case study used for WSOL [Pat03a, Pat03b]. The experiments measured additional overhead of WSOI when performing monitoring of WSOL service offerings. The overhead was measured in terms of average response time and average Java Virtual Machine (JVM) memory usage. Data for WSOI hosting a Web Service and monitoring a WSOL service offering were compared with data for Axis hosting a Web Service and performing no monitoring.

Table 5.1 describes one representative experiment, while Table 5.2 shows its results [Tos04a, MaW04]. In this experiment, the consumer and the provider (a stock notification Web Service) executed on different computers in a local network. When Axis was used, no WSOL constraint was evaluated. When WSOI was used, the provider evaluated a pre-condition, a post-condition, and a response time QoS constraint. The presented values are results of averaging 1000 measurements. These results show that the use of WSOI increased response time 15% and memory usage 4%, compared to response time and memory usage when only Axis was used. In my opinion, this would be acceptable for many consumers and many circumstances. Consequently, many Web Services that now use Axis can use WSOI. When Web Services are distributed over the Internet, the network delay is higher, so the relative WSOI increase of the total response time is lower.

Table 5.1 Description of an Experiment Measuring Additional Overhead of WSOI When Performing Monitoring of WSOL Service Offerings

Description	Axis	WSOI
Number of Web Services	2 (consumer and provider)	2 (consumer and provider)
Distribution	Different computers in a local network	Different computers in a local network
Provider Web Services	Simple stock notification Web Service	Simple stock notification Web Service
Number of evaluated constraints	0	3 (1 precondition, 1 post-condition, 1 QoS constraint)
Number of exchanged SOAP messages	2 (1 request and 1 reply)	2 (1 request and 1 reply)
Start time for measuring response time	Consumer sends the SOAP request message	Consumer sends the SOAP request message
Stop time for measuring response time	Consumer receives the SOAP reply message	Consumer receives the SOAP reply message
How was the average response time calculated?	1000 tests were run, then the average was calculated	1000 tests were run, then the average was calculated
Software participating in provider-side JVM memory usage	Tomcat, standard provider-side Axis modules, Java implementation of the provider	Tomcat, standard provider-side Axis modules, *WSOISessionHandler*, other WSOI-specific modules (version 1), Java implementation of the provider
How was the average JVM memory usage calculated?	For 1000 continuous test runs, JVM memory usage was periodically measured every 20 ms, then the average was calculated	For 1000 continuous test runs, JVM memory usage was periodically measured every 20 ms, then the average was calculated

Table 5.2 Results of the Experiment Measuring Additional Overhead of WSOI When Performing Monitoring of WSOL Service Offerings

Measured Value [Units]	Axis (= A)	WSOI (= B)	Difference (= B-A)	Relative Difference (= (B-A)/A) [%]
Response time [ms]	140	161	21	15.00 %
JVM memory usage [MB]	5.85	6.10	0.25	4.27 %

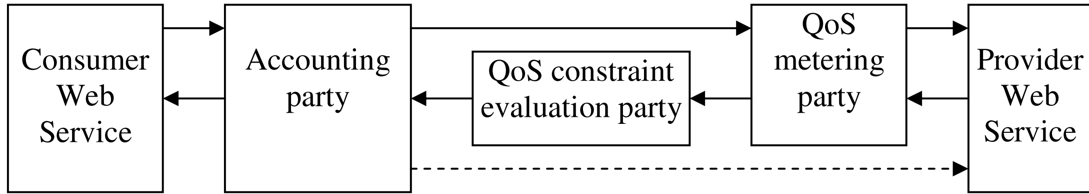


Figure 5.4 An Example Configuration of Management Third Parties as SOAP Intermediaries

5.4 Monitoring of WSOL Service Offerings Using Management Third Parties

Up to now, WSOI examples assumed that provider performs all monitoring activities. However, management third parties or consumer can also use WSOI to perform Web Service monitoring activities. Figure 5.4 shows an example configuration of management third parties as SOAP intermediaries. In this example, only QoS constraints are evaluated and distinction is made between the accounting party, the QoS metering party, and the QoS constraint evaluation party. (The evaluated QoS constraint is the same as in the previously discussed example in Figure 5.3 where all WSOL-related monitoring was done inside the provider Web Service.) The arrows represent passing of information and control between different management parties, achieved through sending of SOAP messages. The order of management parties for a particular context is described in WSOD files and internally stored in WSOI data structures. The technical details of the implementation of such exchange of SOAP messages are in [MaW04], so I just give a high-level overview here. When a consumer submits a request for executing a provider's operation, the management third parties are organized as SOAP intermediaries for the request, as well as the potential response message. The request from the consumer to the provider first goes through the accounting party, which logs that the request has been made. The request is then forwarded to the QoS metering party, which performs necessary activities. For ex-

ample, for metering response time the QoS metering party logs the time that will be considered as the beginning time of the operation invocation. Next, the request message is forwarded to the provider Web Service, which performs the requested operation and sends the response message. The response message first goes through the QoS metering party. In the response time measurement example, the QoS metering party logs the time that will be considered as the ending time of the operation invocation and subtracts from it the logged beginning time. The information about the measured QoS metrics is added into a SOAP header of the original response message from the provider and thus transferred to the QoS constraint evaluation party. The QoS constraint evaluation party receives the response message and the information about the measured QoS metrics and evaluates appropriate constraints. It forwards to the accounting party the response message together with the information whether the evaluated constraints were satisfied or not and appropriate details if some constraints were violated. The accounting party logs the received management information, calculates prices and/or penalties to be paid, and forwards the response message and appropriate management information to the consumer. It can also notify the provider with appropriate management information details, particularly if some QoS constraints were not satisfied. This can help the provider to adapt its behavior to meet guarantees for future operation requests. The additional SOAP message from the accounting party to the provider shown in Figure 5.4 is optional. This is because there are a couple of other ways to transport management information from the accounting party back to the provider. One of them is piggybacking this information onto the next SOAP message from the consumer to the provider, while another is periodic sending of reports.

Some QoS metrics, such as availability, can be measured using probing instead of message interception. As discussed in Section 3.4, a probe is be specified in an attribute of the <*operationCall*> WSOL element and can provide its results to a constraint evaluation party in a push or pull way. When the called operation is specific to the probe (i.e., not standardized), then it is not implemented by WSOI. In fact, the probe need not even implement WSOI in this case. However, the called operation can be one of the operations in the *SONotification* SOM port type and then it is implemented by probe's WSOI. Probes cannot piggyback management information into SOAP headers, as SOAP intermediaries can. On the other hand, SOAP intermediaries can exchange WSOL-related management information using operations of the *SONotification* SOM port type or external operation calls in WSOL constraints.

Probing can have lower run-time overhead than message interception. However, there are several issues with probing in a business-to-business (B2B) environment. Most importantly, it is easy to implement providers in such way that they give excellent QoS to probes, while the actual QoS that real consumers get is significantly lower. Even when providers are not deceiving, a consumer is often not interested in some average QoS (determined by a probe), but the QoS it particularly gets. Due to these reasons, I focused the development of the WSOI prototype on SOAP message interception, while a full prototype implementation of a WSOI management third party that acts as a probe can be done in the future.

Management third parties are important to achieve manageability, flexibility, and trust. Further, the delegation of Web Service monitoring activities to management third parties reduces the run-time overhead placed on the provider Web Service and its consumers.

However, this delegation also introduces additional compositional complexity and increases the overhead, including delays, of communication between the involved parties. Due to these reasons, the number of third parties should not be very high. These two opposite sets of issues can be balanced by concentrating monitoring and management actions into one or few management third parties. For example, replacing the three separate management third parties for accounting, QoS metering, and evaluation of QoS constraints shown in Figure 5.4 with only one management third party for all three functions reduces the overhead.

5.5 WSOI Modules Implementing Dynamic Manipulation of Service Offerings

WSOI implements the algorithms and protocols for dynamic manipulation of service offerings described in Section 4.2, with:

1. WSOI-specific data structures,
2. modules for Service Offering Management (SOM) port types,
3. the *SOMgmtDecisions* module, and
4. *CInitSwitchSOThread* and *PrInitSwitchSOThread* modules.

Unlike the modules described in Section 5.3, these WSOI modules are not based on the standard modules provided in Apache Axis. I designed and implemented them in the WSOI prototype.

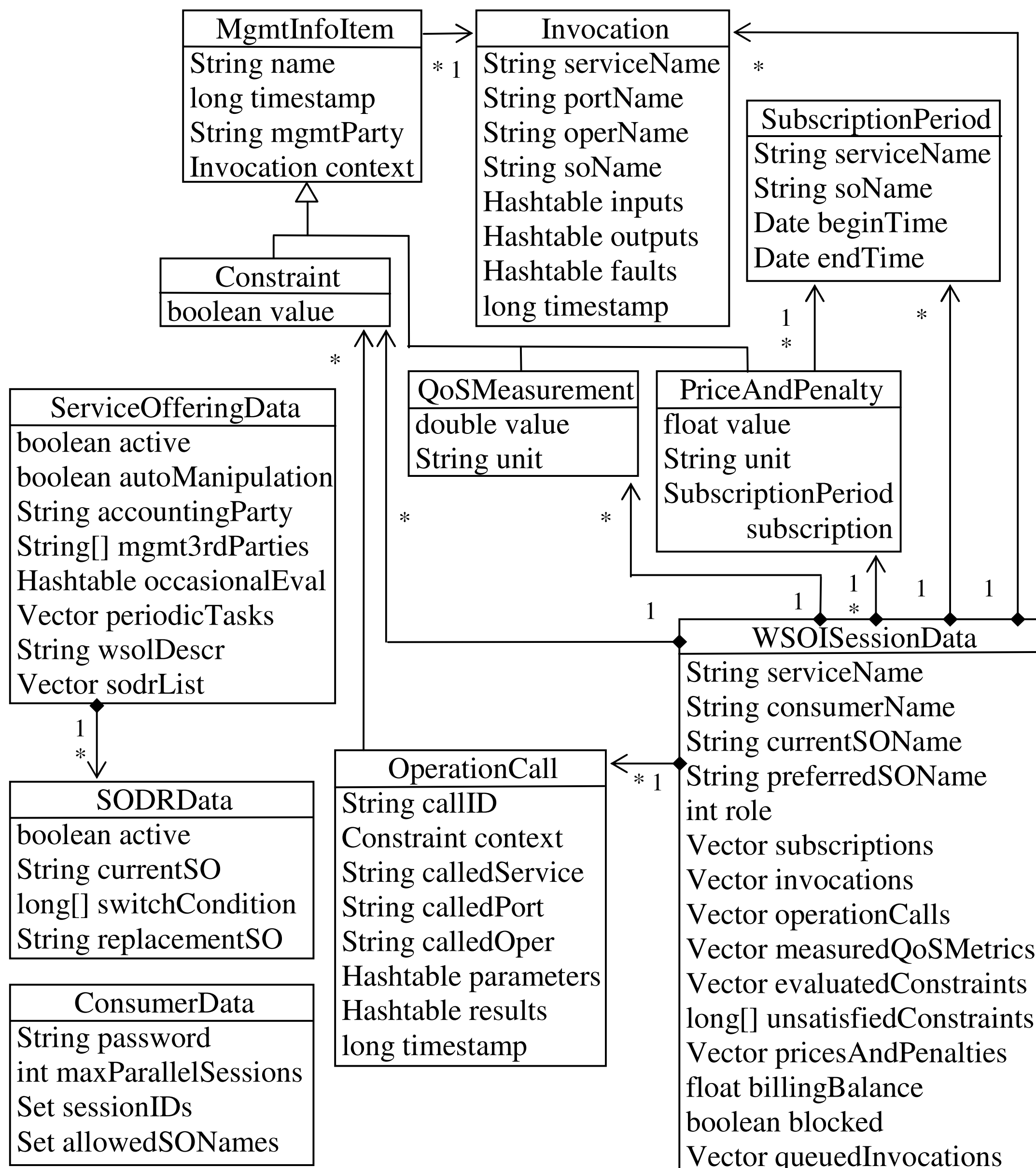


Figure 5.5 Partial UML Class Diagram of the WSOI Management Information Model

MessageContext properties only store monitoring information for the latest invocation. Therefore, WSOI additionally stores descriptions of WSOL constructs and the actual measured or computed data into several **WSOI-specific data structures**. Most WSOI-specific data structures are used for both monitoring and manipulation of WSOL service

offerings and are updated by WSOI-specific handlers. They store information used for accounting of executed operations and evaluated WSOL constraints. Additionally, these data structures store information essential for determining whether manipulation of service offerings is necessary and what is the appropriate replacement service offering.

Figure 5.5 shows the most important WSOI classes that store WSOI-related data. All shown data members in these classes are private. The public operations that manipulate these data members are not shown in this figure due to the space limits.

An instance of the ***ServiceOfferingData*** class stores description of one service offering, its current activity, and data about its service offerings dynamic relationships. It keeps the value of the '*autoManipulation*' attribute of the service offering, the information about the accounting party and all management third parties, the information about occasional and periodic monitoring activities, and, possibly, the complete WSOL description. The data about service offerings dynamic relationships is in instances of the *SODRData* class. While this class fully supports the older format for service offerings dynamic relationships, the evaluation of expressions used in the newer format can be coded into its subclasses.

An instance of the ***WSOISessionData*** class contains important session-related information. In particular, it stores the name of the consumer, the provider Web Service, the currently used service offering in this session, and, possibly, the preferred service offering that is currently deactivated. It also keeps information whether the management party that uses it plays the role of a consumer, a provider, an accounting party, and/or a party that performs some monitoring or evaluation. In some circumstances, different roles perform different actions. For example, only accounting parties block and queue requests

from consumers when switching is in progress. Further, run-time monitoring information is also archived here. For this, data members of this class keep history of subscriptions, invoked operations, operation calls performed in constraints, QoS measurements, evaluated constraints, and calculated prices/penalties. This is stored in instances of classes *SubscriptionPeriod*, *Invocation*, *OperationCall*, *QoSMeasurement*, *Constraint*, and *PriceAndPenalty*, respectively. The latter three are subclasses of *MgmtInfoItem*. Accounting parties and, possibly, providers and consumers keep this history for all activities in the session, while management third parties usually keep only the history of their operation. The list of unsatisfied constraints and the billing balance for the session (the sum of all prices and penalties to be paid) are also determined and stored here.

An instance of the ***ConsumerData*** class stores necessary information about a consumer, such as the password used for identification to open a session, information how many sessions the consumer can open in parallel, the list of open sessions, and the list of service offerings the consumer is allowed to use. This information is stored only on the provider side.

Figure 5.2 shows that one of the main modules in WSOI is the *WSOIDataHolder* class. For every hosted Web Service, an instance of this class contains several hashtables with information necessary for WSOI monitoring and management activities. The ***contextHandlersTable*** **hashtable** [MaW04] stores one *WSODChainContainer* instance for every context described with a *Request* object. A *WSODChainContainer* instance contains WSOI-specific handlers that *WSOIChain* will use for processing of the request and response message in this context. The ***soTable*** **hashtable** stores one *ServiceOfferingData* instance for every used service offering name. The ***sessionTable*** **hashtable** stores one

WSOISessionData instance for every session ID. The ***consumerTable*** **hashtable** stores one *ConsumerData* instance for every consumer name. The ***uriTable*** **hashtable** stores URIs used by the involved management parties. Its keys are *String* objects that contain concatenations of party names and used port type names (e.g., for SOM port types), while the values are the corresponding URIs of these ports.

I implemented, tested, and documented all these data structures and integrated them into the WSOI prototype.

To be able to participate in the protocols for manipulation of service offerings, management parties have to implement and expose appropriate operations from the SOM port types *SOM_Prov*, *SOM_MgmtP*, *SOM_AccP*, *SOM_Cons*, *SODRMgmt*, and *SOSecurity*, described in Section 4.3. The other SOM port types are used for monitoring of service offerings. WSOI implements operations from SOM port types and exposes them to other management parties and, possibly, external management entities through management ports. In this way, there is no need for hosted Web Services to implement them individually. Further, the implementation of management operations can be shared by different Web Services using the same WSOI instance.

The current WSOI prototype contains implementations of SOM operations shown in Figure 4.3, as well as several others. Substantial amount of code in these modules was automatically generated using Java2WSDL and WSDL2Java tools provided by Axis. However, I wrote the crucial code that performs WSOI-specific activities. The level of implementation varies. Most importantly, the implementation of operations crucial for dynamic manipulation of service offerings is relatively complete, tested, and documented. While the actual core implementation of these operations is in *CInitSwitch-*

SOThread, *PrInitSwitchSOThread*, and *SOMgmtDecisons* modules, several other modules check input parameters for these operations, call appropriate implementations, and generate outputs or potential exception messages. On the other side of the spectrum, implementations of operations from SOM port types *SOInfo* and *SOComparisons*, along with some other less important operations, only check input parameters and initial conditions and return a fictional result or throw an exception.

To enable execution of consumer- and provider-initiated switching in separate threads, the provider-side implementation of these protocols is in *CInitSwitchSOThread* and *PrInitSwitchSOThread* classes. These classes are subclasses of Java's *Thread* class, so they can execute in a separate thread or inside an existing thread. The protocol implementations follow the description in Section 4.2 and Appendix A, but they are not modularized (i.e., they contain optimizations) and do not yet support relaying of consumer requests between accounting parties. That is, when an accounting party receives a switching request, it blocks further requests in this session and if consumer submits any request in the meantime, the accounting party returns a fault message.

SOMgmtDecisions module implements the other algorithms and protocols for dynamic manipulation, as well as several auxiliary operations that are used in these algorithms and protocols. Examples of these auxiliary operations are '*checkSwitch()*' and '*block()*' from Figure 4.1. The operations of the *SOMgmtDecisions* module use the WSOI-specific data structures and classes implementing these data structures provide operations that are used for management decisions.

The implementations of the auxiliary operations, and the algorithms for deactivation, reactivation, and deletion of service offerings are complete, tested, and documented. Lim-

ited support for creation of service offerings is implemented to enable scenarios based on the re-use, re-parameterization, and re-ordering of WSOI-specific handlers, as discussed in Subsection 4.2.6. The current implementation of the '*createSO()*' operation expects as input a WSOD file and some other information that can be generated by a WSOL compiler and updates the appropriate *WSOIDataHolder* instance (e.g., the *contextHandlersTable* within it) with this information. In this way, the existing WSOI-specific handlers can be used in new contexts. The current WSOI prototype provides very limited support for the manipulation of SODRs and allowing/disallowing consumers to use a service offering, mainly through auxiliary operations that are also used for other purposes.

I now illustrate how the presented WSOI modules for dynamic manipulation of service offerings are used together for the provider-initiated switching scenario discussed in Section 4.2.3. The independent accounting party uses the '*inform()*' operation from the *SONotification* port type to inform the provider about the measured QoS metrics, evaluated constraints, and calculated prices/penalties. The provider-side implementation of this operation updates the *WSOISessionData* instance for the current session. When the received information about evaluated constraints is added to the *WSOISessionData* instance, its information about the history of unsatisfied constraints is also updated automatically. If there were unsatisfied constraints in the latest invocation, the *WSOIDataHolder soTable* hashtable is used to find the *ServiceOfferingData* instance for the used service offering. Then, the '*getReplacementSOName()*' operation of *ServiceOfferingData* is used to find the replacement service offering, if any. This operation iterates through all *SODRData* instances stored in this *ServiceOfferingData* instance and determines whether they are active and their switching condition is satisfied. If a match is found, the '*check-*

Switch()' operation of the *SOMgmtDecisions* is invoked to determine whether the found service offering is active (the information is stored in the *ServiceOfferingData* instance) and the consumer has the right to use it (the information is stored in the *ConsumerData* instance). The '*getReplacementSOName()*' operation returns either the first replacement service offering that satisfies all conditions or '*null*'. If a replacement service offering is found, the *PrInitSwitchSOThread* module is invoked to perform the protocol for provider-initiated switching. The protocol includes sending SOAP messages to SOM ports of the other parties and, thus, invokes the corresponding WSOI modules that implement these operations.

As discussed in Section 4.5, the WSOI prototype was used as an experimental tool and environment for experiments comparing switching between service offerings and switching between Web Services. In particular, the prototype implementation of modules discussed in this section was crucial for these experiments because it is the only implementation of the mechanisms for dynamic manipulation of WSOL service offerings and more generally, classes of service for Web Services. Without it, Wei Ma and I would not have been able to perform these experiments and my research would have been limited to analytical studies.

5.6 Comparison of WSOI with Related Work

Several recent papers, products, and standardization activities study some kind of monitoring of Web Services, so they are related to WSOI. Several such infrastructures are even based on Apache Axis or its ancestor Apache SOAP Toolkit. However, no related work addresses dynamic manipulation of classes of service for Web Services and the cor-

responding infrastructure support. As discussed in Section 3.6, the closest languages to WSOL are WSLA and WSML. Since these languages are both oriented towards management applications in inter-enterprise scenarios and accompanied by management infrastructures, these infrastructures are the main related work to WSOI.

Management infrastructures for WSLA

The **WSLA framework** [Kel03, Dan02, Deb03, Lud03] contains several modules for run-time negotiation and creation, deployment, execution and compliance monitoring, and termination of SLAs. These are the Establishment Service, the Deployment Service, the Measurement Service, the Condition Evaluation Service, the Management Service, and the Business Entity.

The Establishment Service supports negotiation and authoring of SLAs. It seems that it requires human involvement (particularly of system administrators) for these activities. [Deb03] also mentions that negotiation of SLAs can be performed by Business Entities, without human involvement. This module contains business knowledge, goals, and policies of a party and makes decisions about SLOs compliant with its business goals. The Deployment Service validates the created SLA and distributes the whole SLA or only appropriate parts to management third parties. For this, the provider or the consumer generates a document in the Service Deployment Information (SDI) format, which is a subset of the WSLA language, and distributes it to management third parties that it sponsors. The management third party uses this document to configure itself.

The Measurement Service maintains information about the current system configuration and run-time values of measured QoS metrics and aggregates these values into SLA parameters. The Condition Evaluation Service compares measured or calculated values of

SLA parameters provided by the Measurement Service with SLOs and notifies Management Services of the provider and the consumer about the results. More than one Management Service and Condition Evaluation Service can be used. They can be inside the provider, the consumer, or management third parties (probes or intermediaries).

The Management Service executes corrective actions if SLOs were violated. It can be implemented as a part of an existing systems management platform. To validate the business impact of these corrective actions, the Management Service always consults the Business Entity, which entity approves or rejects the proposed actions. It might even suggest an alternative management action that is more appropriate from the business viewpoint. Since these are complex tasks, they might require input from humans.

These WSLA modules implement several port types with operations for exchange of management information. Examples of these port types are *GetMetricValue*, *ParameterUpdate*, *GetSLAParameterValue*, *Notification*, and *StandardMeasurementIF*. The operations from my *SONotification* SOM port type are similar to these operations.

The **SLA Compliance Monitor** [Dan02] is a Java-based prototype of the WSLA framework. It is a part of the IBM Emerging Technologies Toolkit (ETTK), previously known as the IBM Web Services toolkit (WSTK). It includes a general-purpose Measurement Service, a general-purpose Condition Evaluation Service, a Deployment Service, and WSLA Authoring Service. The implemented Measurement Service supports metric definition with a rich set of functions and includes an extensible set of plug-ins that read measurement data. The Condition Evaluation Service supports a wide range of predicates. The Deployment Service includes a simple WSLA repository and functions for lifecycle management of SLAs. The WSLA Authoring Service is a simple Establishment

Service that enables creation of WSLA templates and filling of these templates. The Management Service and the Business Entity were not implemented. The provider Web Service executes in a servlet engine of a Web application server. The Web application server exposes management information through monitoring and management interfaces, accessed by other modules of the WSLA framework. There can also be a provider-side administration console for interaction with human system administrators.

Overall, the WSLA infrastructure is more powerful, but also more complex, than WSOI. It is assumed that both the provider and the consumer implement the Establishment Service, the Deployment Service, the Management Service, and the Business Entity. While the design of the Business Entity and the Management Services is not elaborated upon, they perform complex tasks, which require considerable infrastructure support. In particular, since selection of WSOL service offerings is simpler than negotiation of SLAs, WSOI avoids the complexity of the WSLA Business Entity. Reusing the existing systems management platforms, at least partially, for the modules in the WSLA framework has both advantages and disadvantages. An advantage is tighter integration with the management of other parts of a complex distributed system and the consequent ability to collect additional relevant management information and make better management decisions. A disadvantage is additional complexity. Contrary to WSOI, the WSLA framework does not contain a separate service for accounting activities. While the WSLA Management Services could be extended to perform such activities, this topic was not discussed. Both the WSLA framework and WSOI support management third parties that act as intermediaries and probes, but WSOI emphasizes SOAP intermediaries, while the WSLA framework better supports probes. When the number of consumers is high, less

memory is needed for storing descriptions of service offerings in WSOI than descriptions of custom-made SLAs in the WSLA framework.

The implementation emphasis in the SLA Compliance Monitor is different from the emphasis in the WSOI prototype. The important part of the WSOI prototype is implementation of algorithms and protocols for manipulation of service offerings, while the SLA Compliance Monitor does not implement autonomous negotiation of SLAs. On the other hand, the SLA Compliance Monitor implements a rich set of functions and predicates. While this makes it reusable for monitoring of many different SLAs, it also enlarges complexity and run-time overhead. With the SLA Compliance Monitor, management third parties know only about their measurement or condition evaluation activities, but contain general code for a wide range of activities. With WSOI, management third parties know only about their activities and contain only WSOI-specific handlers for these activities. There is no publicly available code to compare extensibility of the implemented Measurement Service and Condition Evaluation Service with the extensibility of WSOI, but this might be another advantage of WSOI.

To conclude, WSOI has some advantages over the WSLA framework and its current prototype. Since WSOI addressed several topics that the WSLA framework did not address (or, at least, did not address in such detail), these solutions can be integrated into a future infrastructure for WSLA.

Management infrastructures for WSML

Several papers [vMo02, Sah02b, Mac02] present management infrastructures that use WSML. Most importantly, [Mac02] presents the **Web Services Management Network (WSMN)**, a management architecture for federated management of Web Services from

different management domains, e.g., different businesses. The main elements of the infrastructure are WSMN intermediaries, each acting as a proxy between a Web Service and the outside world. WMSN is a logical overlay network of such intermediaries. WSMN intermediaries monitor and enforce SLAs and exchange management information necessary for configuration of monitoring activities. They communicate using a set of protocols for SLA-based management of relationships between Web Services. [Mac02] distinguishes three classes of protocols: life-cycle, measurement, and assurance (e.g., negotiation) protocols. A WSMN intermediary is embedded in a SOAP router and contains WSMN engines that measure QoS and manage SLAs, modules implementing the WSMN protocols, and applications (e.g., a console) that use management information. While WSMN intermediaries can use SOAP message processing capabilities, they use SOAP headers only to exchange information about connections between SOAP messages (e.g., transactions). Contrary to WSOI, all measures are exchanged using the measurement exchange protocol. WSMN intermediary is similar to the SLA engine presented in [Sah02b]. The latter paper discusses automated SLA monitoring and presents a customizable SLA engine called **Business Management Platform Agent (BMP Agent)** that performs SLA monitoring and evaluation as a proxy that is attached to a SOAP router. The BMP Agent prototype was implemented in Java, over Apache SOAP Toolkit, an ancestor of Apache Axis. WSMN intermediaries and BMP Agents can be implemented at provider and/or consumer side. While [Mac02] and [Sah02b] do not discuss management third parties, [vMo02] mentions that third parties can be added to WSMN. Modules that perform creation and/or negotiation of SLAs are not described.

To illustrate the generality, power, and complexity of a WSMN intermediary, let me describe only one of its parts – the SLA Engine. The main part of the SLA Engine is the Management Process Controller, which receives an SLA, executes a monitoring process flow, and informs the SLA Customizer. SLA Customizer sets alarms at the Event Manager, creates an SLO object, and registers it in the SLA Repository. When triggered, SLO Evaluator evaluates the SLO, using the information stored in a high performance database. If the SLO Evaluator determines a violation, the SLA Violation Engine stores the violation record and triggers appropriate actions. Apart from the SLA Engine, a WSMN intermediary has a number of other modules. For example, the Model Generator model creates a model of Web Services from WSDL and WSFL files and stores this model into the Model Repository. All measurements are attached to this model.

While a WSMN intermediary is more general and more powerful than WSOI, it is also more complex. The main reason why WSOI is simpler than a WSMN intermediary is the use of WSOI-specific handlers and chains to perform monitoring activities. Further, decisions related to what QoS measurements and constraint evaluations to perform and when (e.g., for what operations and service offerings) to perform them are captured in WSOD files. While generation of WSOI-specific handlers and WSOD files by a WSOL compiler takes some time, it is performed only once and subsequent uses of compiled service offerings need not perform such activities. Next, WSOI need not store the full model of monitored Web Services. The complete information about the performed measurements, evaluations, prices, and penalties has to be stored only at the accounting party. There are also several other, less important, differences that lead to a simpler architecture of WSOI.

[Far02]

[Far02] is a survey of Web Service Management (WSM) issues and some possible approaches for their solution. The paper starts with an overview of application management and a short introduction to Web Service technologies. While WSM is discussed from the application management perspective, the authors identify distinctive characteristics of Web Service technologies. They identify three WSM principles that they discuss in detail: 1) the use of separate management interfaces; 2) data collection by the run-time infrastructure, such as SOAP engines; and 3) the use of event collectors – entities (e.g., Web Services) to which various management events are sent. Further, several general Web Service management patterns are discussed. At the end of the paper, the management support of the IBM Web Services Toolkit (WSTK) is examined. It contains Apache Axis SOAP engine and monitors and collects run-time data, e.g., related to availability and average response time. It implements a generic management proxy that works with Java Management Extensions (JMX). While this paper does not discuss Web Service Composition Management (WSCM) issues nor monitoring and manipulation of classes of service for Web Services, it is an important related work to WSOI. WSOI supports some of the WSM principles suggested by [Far02]: SOM port types are separate management interfaces and WSOI is a run-time infrastructure that performs Web Service monitoring and data collection. However, WSOI does not use event collectors. Instead, management information is exchanged through SOAP headers and several operations in SOM port types. WSOI also supports management third parties, which [Far02] does not discuss. I also do not agree with several suggestions by these authors, notably that management interfaces should be published in a special registry, separate from the registry used for

business functionality. I also think that Web Services, not their run-time infrastructure, should expose management interfaces, even when the implementation is actually provided by the run-time infrastructure. This makes the discovery of supported management interfaces easier and uniformly applies to Web Service implementations that already contain code for these management interfaces. While both WSOI and the IBM Web Services Toolkit use Axis as a SOAP engine and perform monitoring of Web Services, the IBM tool collects only a limited variety of data for predefined QoS metrics. Further, it does not examine bodies of SOAP messages, which means that it cannot check functional constraints and access rights nor exchange management information through SOAP headers. The approach built into WSOI is more general and does not depend on JMX.

Monitoring infrastructures for WS-QoS

The main concept in the **monitoring infrastructure for WS-QoS descriptions** [Tia03] is a QoS proxy. It resides between the Web Service level and the transport level and matches QoS parameters between the two levels. This assumes that the transport level supports QoS, e.g., by using DiffServ. This is a serious limiting assumption. Monitoring of QoS is done in the QoS proxy and management information is transported in SOAP headers. A GUI is used to report performance to providers and all interested consumers, irrespective whether they currently use this class of service or not. The idea is to enable a consumer to use this performance data for choosing appropriate class of service.

A similarity between WSOI and this infrastructure is the use of SOAP headers for transport of monitored information. However, there are a number of differences. WSOI deliberately does not address publication of monitored information to other consumers because such publication can be misleading. On the other hand, WSOI contains support

for dynamic adaptation of Web Service compositions using manipulation of WSOL service offerings.

Smartware

Smartware [Shar03] is a Web Services Management infrastructure with solutions that could complement the current WSOI architecture. Similarly to WSOI, Smartware is based on Apache Axis, but the authors focus on differentiated scheduling of incoming requests based on priorities assigned to their contexts. Information about the user, the provider-side application, and the consumer-side device determine the priority of a request context. A consumer sends this information in SOAP headers of the request. On the provider side, Smartware adds an interceptor, a scheduler, and a dispatcher. The interceptor reads request context information from SOAP headers and determines request's priority based on the information in a provider-side XML file. Based on this priority, the scheduler stores the request in one of multiple request queues. Using a configurable scheduling policy, the scheduler fetches a request from one of these queues and passes it to the dispatcher. The dispatcher forwards the request to the provider. In my work, the invoked operation and the used WSOL service offering determine request context. Only a mapping from such request context information to request priorities has to be provided for Smartware to work with WSOI. This would be a significant addition to WSOI. However, the Smartware solutions are only a step forward towards the actual control of the behavior of the provider to meet QoS guarantees. To be able to meet QoS guarantees, a scheduling policy has to take into consideration the values in QoS guarantees, execution times, availability of resources, time spent in request queues, monetary benefits from

meeting guarantees, monetary penalties for not meeting guarantees, and maybe other information. Since this is very complex, a lot of future work in this direction is needed.

OGSI and WS-Agreement

The Grid computing community [Fos02] has the goal to provide capabilities for dynamic, on-demand, Internet-wide sharing of various computing resources: applications, databases, computational power, storage memory, network bandwidth, etc. The idea is to make computing resources as easily accessible as electricity is accessible through the current electric grid. There were several generations of Grid computing architectures, from the Globus Toolkit over **Open Grid Services Architecture (OGSA)** to the recent Web Services Resource Framework (WSRF). Every subsequent generation becomes more closely aligned with XML Web Services. In OGSA [Fos02], diverse computing resources are exposed on a network as Grid Services. A Grid Service is a Web Service that implements a set of predefined port types and follows specific conventions. The predefined port types contain operations for discovery, dynamic creation, lifetime management, notification, and manageability (e.g., monitoring). Very specific conventions are used for naming and upgradeability. There is also some support for authorization and concurrency control. The Open Grid Services Infrastructure (OGSI) implements OGSA. While OGSA and OGSI have some support for management of Grid Services, there are important differences from WSOI. Most importantly, a Grid Service is a very special type of a Web Service, so many (maybe even most) OGSI solutions cannot be applied to other Web Services. Further, management projects within the Grid community predominantly concentrate on resource management or resource reallocation [AlA03] and do not address specification, monitoring, and dynamic manipulation of classes of service for Web Ser-

vices. It is not yet clear how the recent Web Services Resource Framework (WSRF) will relate to the management of general Web Services and their compositions.

The OGSA work that is closest to my research is **WS-Agreement**, previously known as OGSI-Agreement [Cza03]. This work defines several port types for creation and re-negotiation of agreements for Grid Services. In this work, an agreement contains provider's commitments towards its consumer, while consumer's responsibilities have to be negotiated separately. Individual commitments are represented with agreement terms. For example, a term can represent a service level objective. WS-Agreement defines several domain-independent agreement term types expressed with extensions of WS-Policy. Additional domain-specific agreement term types can be defined separately from the WS-Agreement specification. The generality of the defined agreement term types leaves possibilities for incompatibility of languages that will define domain-specific term types. An agreement is represented with an Agreement Grid Service, not as a document. This enabled managing life cycle of agreements through standard OGSI capabilities for monitoring and lifecycle management of Grid Services. Both WS-Agreement port types and my Service Offerings Management (SOM) port types address creation and dynamic adaptation of special types of contract (agreements or classes of service) by defining special operations and port types. WS-Agreement is more abstract and more flexible, while SOM port types are much more detailed and more powerful. SOM port types can influence the WS-Agreement community to add capabilities for dynamic manipulation and comparisons of agreements.

Academic papers researching one management functional area for Web Services

Several other recent academic projects focus on particular management functional areas of Web Service Management (WSM) and Web Service Composition Management (WSCM), often performance (e.g., [Ouy03]) or security (e.g., [Bro03]). However, these management solutions cannot be easily transferred or generalized to the other management areas. In addition, they do not address comprehensive description of Web Services for management activities.

Industrial infrastructures for WSM and WSCM

Due to the importance of Web Service Management and Web Service Composition Management, several companies sell products in this area. I followed this industrial market segment throughout my Ph.D. research. My recent survey is presented in [Tos04c]. The current products can be classified into several categories. At one end of the spectrum are many companies that advertise their existing application and/or system management software, often platform-specific, as suitable for Web Services. Notable examples are Microsoft Application Center and a group of products based on JMX. Similarly, some support for WSM, in the form of separate products and/or enhancements of existing products, is also integrated into the main enterprise management suites: IBM Tivoli, Hewlett-Packard OpenView, Computer Associates Unicenter, and BMC Patrol. Then, several companies developed specialized products for one functional area of WSM, most often performance or security. Some examples of products in this area are Flamenco Networks, Netegrity TransactionMinder, and Reactivity XML Firewall. Further, several companies offer families of products that address more than one functional area of WSM: Actional Web Services Management Platform, AmberPoint family of products, Blue Ti-

tan Software Network Director, Infravio Ensemble, Wakesoft Architecture Platform, and WestGlobal mScape. A frequent approach in the latter two groups is to use a Web Service gateway, hub, or proxy that serves as a single point of control, metering, and management. Finally, a couple of products offer some kind of WSCM: Grand Central Communications Business Services Network and WebMethods Integration Platform. While these diverse products have very diverse characteristics, several common features different from WSOI can be noted. First, human administrators have the central role in operating these products. Second, none of these products implements dynamic adaptation mechanisms similar to those studied in this Ph.D. research and integrated into WSOI. Third, while some of these products (e.g., AmberPoint Service Level Manager, Blue Titan Software Network Director, Infravio Ensemble, and WestGlobal mScape) have the concept of a custom-made contract or an SLA, they do not support classes of service. Fourth, for almost all products, Web forms used by human administrators determine and limit the set of monitored and controlled QoS metrics. Fifth, from the available information it seems that these products (except maybe Infravio Ensemble) do not define new powerful and flexible XML languages for the formal description of SLAs, comparable to WSOL. While they might internally use some XML representations for the information human administrators enter into Web forms for SLAs, these representations have limited expressiveness.

WS-Manageability

An important standardization effort in the area of management of Web Services and their compositions is the work of the **OASIS Web Services Distributed Management (WSDM) Technical Committee (TC)**. Its goal is to define solutions for both distributed

management of Web Service and distributed management of general computing resources using Web Services. One submission to this industrial standardization body **is WS-Manageability** [Pot03], which tries to address the former goal. To enable incremental and modular development of management capabilities, the authors identify management topics, each representing a particular problem or a management domain. Then, they describe using WSDL the functional capability within a topic using three aspects: properties, operations, and events. The WS-Manageability specification discusses five topics: identification, state, configuration, metrics, and relationships. The only topic somewhat related to WSOI is management of metrics. However, WS-Manageability discusses only very general properties, operations, and events. For example, only two operations are defined – one to reset value of metrics and another to start/stop collection of metrics. It seems that WS-Manageability authors assume that the managed Web Services only provide measured values for metrics, while their aggregation, evaluation of conditions, billing, and other management activities are performed by external managers. I do not agree with such an approach. A Web Service is an abstraction that encapsulates the implementation of the business (functional) logic and appropriate software and hardware infrastructure, including SOAP engines. Consequently, manageability and complex management mechanisms can be built into Web Services without affecting the implementation of the business logic, e.g., as it is done with WSOI. Outsourcing manageability to external managers increases the number of exchanged SOAP messages, and, thus, delays and run-time overheads. In my opinion, much more can be standardized for management of Web Services than is present in WS-Manageability. Partial solutions for several issues, such as representation of exchanged management information (including QoS measurements,

evaluated constraints, and prices/penalties) and definition of management interfaces (like SOM port types), are built into WSOI and can be built upon in future standardization activities.

Business process management tools

In the information systems community, there has been a lot of work on **tools for business process management (BPM)**. [Lay02] gives an overview how Web Services relate to business processes and their re-engineering and management. When a business process is implemented with a Web Service composition, business process re-engineering is implemented with dynamic adaptation of this Web Service composition. Business process management achieves monitoring of business processes, as well as enactment of business process re-engineering results. It can include management of contracts between participants. However, business process management tools traditionally do not automate management – they let users visually compose business processes, monitor execution of these processes, present results on management dashboards, and then let humans perform management actions. There are several important differences between WSOI and business process management tools. WSOI is not intended for human users, so it has no visual tools. Next, it does not directly support dynamic service composition and re-composition. Further, a business process management tool monitors complete Web Service compositions, while WSOI monitors consumer-provider pairs. On the other hand, the distinctive advantage of WSOI is that it automates management activities, particularly using the mechanisms for manipulation of WSOL service offerings.

Monitoring infrastructure for CBabel

In Section 3.6, I examined similarities and differences between WSOL and the CBabel Architecture Description Language for software components [Cerq03]. The authors of CBabel also designed a supporting infrastructure that contains a QoSAgent that monitors and controls QoS, a Configurator reflective middleware for configuration management, a Contract Manager that interprets contract descriptions, and an Interactor component that performs a number of management activities and connects the other components. While there are some similarities between the monitoring and dynamic adaptation activities performed by WSOI and the CBabel supporting infrastructure, the actual implementation is different. While [Cerq03] does not provide enough details for such comparisons, it seems to me that the WSOI approach is more efficient. In addition, WSOL is significantly more powerful and detailed than CBabel and this is reflected in the supporting infrastructures. Last but not the least, the CBabel work is not for Web Services.

Summary of the comparisons

Compared to the related works, the main distinctive characteristics of WSOI are the implementation of algorithms and protocols for manipulation of service offerings, the definition and implementation of SOM port types, implementation of WSOL-related monitoring activities in WSOI-specific handlers, support for management third parties acting as SOAP intermediaries or probes, and the management information model that stores WSOL descriptions and run-time management information. Additional good characteristics of WSOI are extensibility, relative simplicity, and relatively low additional overhead (even for relatively simple Web Services) over a widely used SOAP engine – Apache Axis. These and other solutions integrated into WSOI can be reused and built

upon in future commercial products, open-source projects (notably, future versions of Apache Axis), standardization activities, and academic research projects in the areas of Web Service Management and Web Service Composition Management.

6 Conclusions and Future Work

As stated in Chapter 1, the topic of this Ph.D. dissertation was the specification, monitoring, and manipulation of classes of service for XML Web Services. My work answered several research questions related to this topic. It showed why the specification, monitoring, and manipulation of classes of service (service offerings) are useful for Web Services and how to achieve them using the Web Service Offerings Language (WSOL), the Web Service Offerings Infrastructure (WSOI), and the developed algorithms and protocols.

In this final chapter, I first summarize the answers to the research questions stated in Section 1.5, the contributions of this Ph.D. research, the importance of this research, and the used validation methods. In Section 6.2, I explore wider implications of this research, e.g., how it applies to general service-oriented computing that need not be based on XML Web Service technologies. I overview some items for future work in Section 6.3.

6.1 Summary of Contributions and Their Importance

In Section 1.5, I stated that I would answer a number of **research questions**. Let me now restate them, along with a summary of the answers provided in this Ph.D. dissertation:

1. Why are classes of service (service offerings) useful for Web Services? => Service offerings are useful for Web Services because they provide comprehensive description of Web Services and differentiation of service and QoS, address diverse consumers and execution circumstances, and enable balancing of resource utilization and price/performance ratios.

2. What are the main alternatives to service offerings? => The main alternatives to service offerings are contracts, Service Level Agreements (SLAs), consumer profiles, separate ports, and separate Web Services.

3. Why are service offerings more appropriate than alternatives in some circumstances? => In some circumstances, service offerings are more appropriate than alternatives because of the relative simplicity and low overhead (lightweightness) of their specification, monitoring and management, as well as easier, faster, and more lightweight dynamic manipulation.

4. What is needed to enable specification, monitoring, and manipulation of service offerings? => To enable specification, monitoring, and manipulation of service offerings a specification language, a monitoring infrastructure, algorithms and protocols for manipulation, and a manipulation infrastructure have to be developed.

5. What concepts and features should a specification language for service offerings have? => According to my studies, the most important features for a specification language for service offerings are those that were integrated in WSOL and explained in detail in Chapter 3. Many of these concepts and features are original to WSOL.

6. What could be other theoretical contributions of such a specification language, in addition to the support for service offerings? => For example, WSOL enables specification of different static and dynamic relationships between service offerings, supports different types of constraint and statement in one language, fosters reusability of specifications, introduces the concepts of a future-condition and occasional evaluation of QoS constraints, etc.

7. How could monitoring of service offerings be achieved? => Monitoring of service offerings can be achieved in several ways. For WSOI, I chose to extend Apache Axis, a popular SOAP engine, with special modules in a way described in detail in Chapter 5.

8. What mechanisms can be used for exchange of run-time values of management information between management parties? => For exchange of run-time values of management information between management parties, piggybacking management information into SOAP headers and invocation of special operations, such as those in the *SONotification* SOM port type, can be used. The former is a lightweight solution that can be used for all management parties except third party probes, while the latter has higher overhead but can be used by all management parties.

9. Is the overhead of monitoring of service offerings acceptable? => According to the experiments comparing response time and JVM memory usage of Axis and WSOI, this overhead can be sufficiently low for many uses of Web Services.

10. What mechanisms can be used for manipulation of service offerings? => For manipulation of service offerings one can use consumer-initiated switching, provider-initiated switching, deactivation, reactivation, deletion, creation, allowing use, and disallowing use. Additionally, manipulation (creation, deletion, deactivation, and reactivation) of dynamic relationships between service offerings can be used.

11. What are the precise algorithms and protocols for these mechanisms? => The precise algorithms and protocols for manipulation of service offerings were summarized in Section 4.2 and built into WSOI.

12. How can these algorithms and protocols be implemented? => My implementation of these algorithms and protocols is in WSOI's *SOMgmtDecisions* module, *PrInitSwitch-*

SOThread, *CInitSwitchSOThread*, modules implementing SOM port types, and WSOI-specific data structures.

13. What data structures are useful for storing information related to monitoring and manipulation of service offerings? => Data structures in the WSOI management information model described in Chapter 5 store information about measured QoS metrics, evaluated constraints, calculated prices/penalties, invoked operations, subscription periods, external operation calls, consumers, and open sessions, as well as descriptions of service offerings and service offerings dynamic relationships.

14. What externally available operations are needed or useful for monitoring and manipulation of service offerings? => The most important of these operations are in SOM port types described detail in Section 4.3.

15. What are the main alternatives to the mechanisms for manipulation of service offerings? => The main alternatives to manipulation of service offerings are re-composition of Web Services, switching between Web Services, and re-negotiation of SLAs.

16. What methodology can be used for analytical comparisons of different dynamic adaptation mechanisms for Web Services? => The number of exchanged SOAP messages gives an approximate measure of simplicity of a dynamic adaptation mechanism and introduced delays, so it can be used as a basis for such analytical comparisons.

17. Which factors influence the delay introduced by various dynamic adaptation mechanisms for Web Services and how? => The derived analytical formulas presented in Section 4.4 show to what extent the number of management third parties, the use of an independent accounting party, and other factors influence the delay introduced by various dynamic adaptation mechanisms.

18. Why are the mechanisms for manipulation of service offerings useful and more appropriate than alternatives in some circumstances? => The analytical studies and experiments comparing switching of service offerings and switching of Web Services and the analytical studies comparing manipulation of service offerings and re-negotiation of SLAs for Web Services presented in Chapter 4 show that manipulation of service offerings is, in principle, simpler, faster, and with lower run-time overhead.

19. What are the limitations of these mechanisms? => The main limitations of manipulation of service offerings are that service offerings differ only in constraints and/or management statements, which may not be big enough difference, and that appropriate service offerings may not exist or be created.

20. What is the recommendation about using service offerings for Web Services and the mechanisms for their manipulation? => I view service offerings as a complement to, but not a complete replacement for, the alternatives. An ideal Web Service Management and Web Service Composition Management system would combine service offerings and the mechanisms for their manipulation with more powerful alternatives, so that the most appropriate tool can be used for a particular context.

Summary of contributions

To summarize, the main **contributions** of this Ph.D. research are:

1. The concept of using classes of service for Web Services. The analysis of their benefits and limitations. The analytical comparison with alternatives, such as SLAs and profiles. The argumentation that, under certain conditions, classes of service can be more appropriate for Web Services than alternatives.

2. The development of the Web Service Offerings Language (WSOL) for the explicit, formal, and precise specification of classes of service, various types of constraint, and management statements for Web Services. The integration into WSOL the concept of a service offerings dynamic relationship and a broad set of reusability constructs to model dynamic and static relationships between classes of service, respectively. The development of the other WSOL features with unique expressive capabilities, relatively low run-time overhead, and support for management applications. Comparative analysis with related languages.

3. The proposal of mechanisms for the manipulation of classes of service for Web Services and creation of appropriate algorithms and protocols. The definition of SOM port types. The analytical and experimental comparison of these mechanisms with alternatives, such as switching between Web Services. The suggestion of integration of the mechanisms for specification, monitoring, and manipulation of classes of service with alternatives.

4. The design of the Web Service Offerings Infrastructure (WSOI) to enable monitoring of WSOL-enabled Web Services, manipulation of classes of service in Web Service compositions, and research of both WSOL and the mechanisms for manipulation of classes of service. The implementation of algorithms and protocols for manipulation of service offerings, as well as used SOM port types. The development of the management information model that stores WSOL descriptions and run-time management information in WSOI. The experiments with WSOI to demonstrate feasibility and usefulness of WSOL and the mechanisms for manipulation of classes of service. Comparative analysis with related management infrastructures for Web Services.

Used validation methods

To validate the work presented in this Ph.D. dissertation, I used several diverse **validation methods**: analyses, comparative analyses, analogies, proof-of-concept prototypes, case studies, analytical modeling, experiments, and peer feedback. In particular:

1. To determine that the concept of a class of service is beneficial for Web Services, I used analysis of benefits and limitations and analogies with other domains. While analogies are not a deterministic and strong 'proof', they are useful indicators. These indications were confirmed by the validation methods listed below.

2. For the study of how classes of service relate to alternatives, I used comparative analysis.

3. The implementability of the WSOL concepts and the syntax correctness of the WSOL grammar were checked through Kruti Patel's development and implementation of the 'Premier' WSOL parser. The semantics and pragmatics of the WSOL concepts and features were, to some extent, tested on a number of examples from the '*buyStock*' case study. Some of these test examples were semantically correct, while the others contained deliberately introduced semantic errors.

4. The implementability of the WSOI architecture and its applicability for the management of Web Services and their compositions were checked through the development of the proof-of-concept WSOI prototype, implemented by Wei Ma and me, and the execution of several test scenarios.

5. The applicability of WSOL for management activities was validated through the development of WSOI and, particularly, its prototype.

6. The implementability of the mechanisms for manipulation of classes of service and the correctness of the corresponding algorithms and protocols were also confirmed through the design of WSOI, the implementation of its prototype, and the execution of various experimental scenarios.

7. The proposed manipulation mechanisms were compared with alternatives using analyses of the number of exchanged SOAP messages and experiments with the WSOI prototype.

8. The analytical formulas generated for switching between service offerings and switching between Web Services were validated on a number of test cases – UML sequence diagrams.

9. The theoretical novelty of the solutions presented in this Ph.D. dissertation was confirmed using a comparative analysis with the related work.

10. As one of the ways to confirm that my Ph.D. research topic is important, I used the feedback (e.g., anonymous peer review) from the international research community, both industrial and academic. The research community recognized the validity of this research problem and my solutions by publishing and citing my peer-reviewed papers. In this area, I published 8 international journal or conference papers that were fully reviewed by anonymous experts, 4 international conference papers that were reviewed based on extended abstracts, and numerous presentations and research reports. Several of these papers were recognized as being among the best at the respective venues. On the other hand, my papers were referenced in about a dozen peer-reviewed papers by other authors, at least two Ph.D. dissertations, and many non-refereed presentations and technical re-

ports. In addition, the research community recognized the validity of the broader research area by working on similar topics.

The importance of the presented Ph.D. research

This Ph.D. research and its contributions are **important** because:

1. In the near future, Web Services will probably be an important, and probably the dominant, distributed computing technology.

2. Monitoring and management are necessary for many business applications of Web Services.

3. This is the first and most thorough work on the specification, monitoring, and management of classes of service for Web Services.

4. Classes of service contain important management information that comprehensively describes and differentiates service and QoS. Compared to alternatives, they are simpler and more lightweight.

5. Compared to related languages, WSOL has a number of unique concepts and features and their combination resulted in unique expressive capabilities, relatively low runtime overhead, and support for management applications.

6. Dynamic manipulation of classes of service is a relatively simple, fast, and lightweight complement to its alternatives.

7. WSOI is the only management infrastructure that supports manipulation of classes of service for Web Services. In addition, its solutions for monitoring of service and QoS of Web Services have several good characteristics, such as flexibility, extensibility, support for SOAP intermediaries and probes, and relatively low additional overhead.

8. The WSOI prototype demonstrates that the suggested solutions are feasible, practical, and useful even for relatively simple Web Services.

9. The results of this Ph.D. dissertation can be reused in future standards and industrial products. WSOL can be used (along with WSLA, WSML, WS-Policy, and several other recent works) as a basis for a future standard for comprehensive description of characteristics of Web Services that cannot be described with WSDL. The original WSOL concepts and features can also be integrated into the related languages. Further, the proposed mechanisms for the manipulation of service offerings can be integrated with their alternatives into a more versatile system for the management and dynamic adaptation of Web Service compositions. The operations in SOM port types can be reused in future standards defining management interfaces of Web Services. The WSOI extensions of Apache Axis could be integrated into future versions of this open-source software, while similar concepts could be reused in other management infrastructures.

6.2 Possible Wider Implications – Beyond XML Web Service Technologies

In several early papers (e.g., [Tos01]), I defined and discussed a '**service component**' as a composable, reusable, and replaceable self-contained unit of service provisioning and management that encapsulates some service functionality and appropriate data. Here, service functionality need not be only software functionality – it can include access to hardware and telecommunications resources. I suggested a service component as an implementation-independent umbrella concept for several service-oriented computing concepts that were used at that time, including the concept of a Web Service. Many other authors note that Web Services are only one possible implementation of the service-oriented ar-

chitecture and that they can be extended to encompass hardware services. Therefore, it is important to examine how the contributions of this Ph.D. research relate to general service components.

The majority of the concepts presented in this Ph.D. dissertation are not dependent on XML and XML Web Service technologies. The WSOL grammar is tightly related to Web Service technologies, but the concepts in WSOL are not. The developed algorithms and protocols for dynamic manipulation of service offerings, as well as the corresponding SOM port types, are not specific to Web Service technologies. While the majority of WSOI modules for service offering monitoring are tightly related to Axis concepts and modules, the modules for manipulation of service offerings are not. To a large extent, WSOI architectural solutions are not dependent on Web Service technologies because such dependencies are mostly encapsulated in the reused Axis modules. The most important exception is that WSOI transports management information in SOAP headers.

Several authors (e.g., [Mil03]) predicted that a future distributed computing platform will integrate and extend technologies currently developed for Web Services, Grid Computing, the Semantic Web, Peer-to-Peer (P2P) systems, pervasive computing, and mobile computing. I believe that in such an environment the importance of classes of service for Web Services and the mechanisms for their manipulation will increase. In the Open Grid Services Architecture (OGSA), Web Services are used not only for representing software, but also as abstractions for hardware and communication resources. For such 'implementations' of Web Services, issues related to QoS, access rights, prices and penalties, resource utilization, and price/performance ratio can be very significant. Since I suggested definition of QoS metrics in ontologies that are outside particular WSOL service offer-

ings, I also see my work as compatible with the Semantic Web technologies. Further, Web Services with classes of service can form peer-to-peer networks. In such a case, manipulation of service offerings enables managing Web Service compositions without an external Web Service Composition Management entity and even without explicit descriptions of Web Service compositions. Since classes of service are a lightweight approach to customization of service and QoS and their manipulation of service offerings is a lightweight approach to dynamic adaptation, I see them as suitable for resource-constrained devices in pervasive and mobile computing. In addition, manipulation of service offerings can be used for handling temporary disturbances, which might occur relatively often in mobile computing. Due to all these reasons, my solutions for the specification, monitoring, and manipulation of classes of service for Web Services can be a basis for the research of the provisioning and management of future distributed computing systems extending XML Web Service technologies.

Self-management is one of the ultimate goals of distributed systems management. It is also the essence of IBM's autonomic computing vision [Kep03] in which computing systems manage themselves, while system administrators provide only high-level objectives. Such a computing system is viewed as analogous to the human autonomic nervous system. The full vision of autonomic computing will be realized in a distant future. However, my work on the algorithms and protocols for dynamic manipulation of service offerings achieves a level of self-management of Web Services and their compositions. Consequently, it can be viewed as a step towards autonomic computing.

6.3 Future Work

While I achieved significant results on the specification, monitoring, and dynamic manipulation of classes of service for Web Services, future research can extend these results and/or apply them to emerging distributed computing technologies.

While WSOL is relatively complete, several items for future work on WSOL can be identified. One example is full compatibility with WSDL version 2.0. Further, WSOL can be extended to achieve compatibility with BPEL4WS. It would be also useful to integrate WSOL with the existing Web Service discovery and selection technologies, such as UDDI. This can be followed by the development and implementation of appropriate WSOL-based Web Service comparison and selection algorithms and heuristics. Apart from such general items for future work, there are several specific ones. For example, partial instantiation of a constraint group template could be added to WSOL. In such an instantiation, values for only some constraint group template parameters would be supplied and the result would be a new constraint group template. A future version of WSOL will also enable that a constraint group template parameter can be replaced with an expression to be evaluated during run-time.

The most important future work item for WSOL is the full implementation of a WSOL compiler to enable automatic generation of WSOI-specific handlers, WSOD files, and other relevant artifacts from WSOL files. Further, the development of WSOL service offerings can be automated or assisted by various other tools, such as tools collecting QoS statistics to define appropriate QoS constraints. In addition, Java Application Programming Interfaces (APIs) for easier generation of grammatically correct WSOL and WSOD files would be useful. For example, such APIs could be used for generation of WSOL and

WSOD files from the contents of internal WSOI data structures after dynamic manipulation (particularly creation) of service offerings. On the other hand, human Web Service administrators can create or modify WSOL service offerings manually, so some visual tools for editing of WSOL and WSOD files would be useful. At present, humans can use general XML tools, such as xmlspy [Alt04] for this purpose.

Additional case studies and, possibly, experiments comparing WSOL and languages based on custom-made SLAs would be useful to better estimate usability and usefulness of these languages and their particular constructs.

While I thoroughly researched the main mechanisms for manipulation of service offerings, I addressed only limited cases of dynamic creation of service offerings. Future study of this complex mechanism seems appropriate. In addition, further research on allowing and disallowing use of service offerings and on manipulation of service offerings dynamic relationships (SODRs) is possible.

The performed analytical and experimental comparisons of the mechanisms for manipulation of service offerings validate the claims of relative simplicity, speed, and low overhead of the proposed mechanisms. Additional analytical studies and experiments with the manipulation of service offerings, and WSOI in general, can be conducted to obtain more precise understanding of benefits and limitations. For example, analytical studies can also encompass factors beyond the number of exchanged SOAP messages. The experiments comparing manipulation of service offerings and re-negotiation of SLAs could also be useful. It is also possible to more thoroughly examine correspondence between results of the derived analytical formulas and the experimental measurements, although the current results show a relatively good correspondence.

WSOI can also be improved and extended in several ways. For example, if additional algorithms and protocols for manipulation of service offerings are developed, WSOI should be extended to support them accordingly. One long-term item for future work is integration into WSOI of the actual control of Web Services to meet the specified constraints. Further, research of how to use the existing distributed system management technologies to collect values for QoS metrics measured in WSOI-specific handlers could lead to an integration of WSOI with these technologies and, consequently, its improved usability. Additionally, WSOL and WSOI could be extended for use with security and privacy technologies for Web Services. For example, different keys could be used for encryption of SOAP message body and various QoS measurements and constraint evaluation results in SOAP message headers, so that only relevant management parties would understands them. However, the latter only improves security and privacy to a limited extent. For full security and privacy of Web Services supporting WSOL and WSOI, additional security and privacy solutions would have to be integrated.

While the previous tasks extend the design of WSOI, several parts of the existing design have to be implemented into a future WSOI prototype: the code of those SOM operations that are not yet fully implemented, the *Timer* module, and the support for occasional evaluation of QoS constraints. It will also be useful to fully implement at least one WSOI management third party that acts as a probe and perform some experiments with it.

Experiments and/or case studies comparing WSOI and tools for languages using custom-made SLAs can also be conducted, but they have to be designed very carefully in order to produce fair and accurate results.

Several promising research projects can be conducted as application and extension of this Ph.D. research to emerging distributed computing technologies mentioned in the previous section. I am particularly interested in applying and extending my research results to Web Services executing in mobile and embedded computing systems, so called m-Services. Some of the specifics of their execution environments that management infrastructures must consider are slow wireless links, scarce memory, and possible occurrence of relatively frequent changes and disturbances. The existing solutions for management of Web Services in non-mobile and non-embedded environments and the current systems for hosting of m-Services do not address these specifics. On the other hand, the relative speed and low memory overhead of my solutions are very important in such execution environments. Therefore, I plan to expand my Ph.D. research into a management system for m-Services and their compositions.

References

Notes: This list follows the formatting guidelines of Lecture Notes in Computer Science (LNCS) printed by Springer Verlag. All given Internet links were valid on June 1, 2004.

Ann.=Annual; Conf.=Conference; Int.=International; Jour.=Journal; Mag.=Magazine; No.=Number; Publ.=Published; Proc.=Proceedings; Res.Rep.=Research Report; Symp.= Symposium; Trans.=Transactions; Ver.=Version; Vol.=Volume; Wsh=Workshop.

[Aim00] Aimoto, T., Miyake, S.: Overview of DiffServ Technology: Its Mechanisms and Implementation. IEICE Trans. Inf. & Syst., Vol. E83-D, No. 5 (May 2000). IEICE (2000) 957-964

[Ais02] Aissi, S., Malu, P., Srinivasan, K.: E-Business Process Modeling: The Next Big Step. Computer, Vol. 35, No. 5 (May 2002). IEEE-CS (2002) 55-62

[AlA03] Al-Ali, R., Hafid, A., Rana, O.F., Walker, D.W.: QoS Adaptation in Service-Oriented Grids. In Proc. of MGC2003 - Int. Wsh. on Middleware for Grid at Middle-ware 2003 (Rio de Janeiro, Brazil, June 2003). On-line at: http://www.wesc.ac.uk/learn/publications/pdf/MGC289_final.pdf (2003)

[Alt04] Altova: XMLSPY 2004. WWW resource. On-line at: http://www.xmlspy.com/products_ide.html (2004)

[Axi02] The Axis Development Team: Axis Architecture Guide, 1.0 Ver. Oct. 7, 2002. The Apache Software Foundation. On-line at: http://archive.apache.org/dist/ws/axis/1_0/xml-axis-10.zip (2002)

[Bak97] Baker, S: CORBA Distributed Objects. Addison-Wesley (1997)

[Bec99] Becker, C., Geihs, K.: Generic QoS Specifications for CORBA. In Proc. of

Kommunikation in Verteilten Systemen - KIVS'99 (Darmstadt, Germany, 1999). Springer-Verlag (1999) 184-195

[Beq03] Bequet, H., Kunnumpurath, M. M., Rhody, S., Tost, A.: Beginning Java Web Services. Wrox. Chapter 3 “Creating Web Services with Java”, pp. 91-132. WWW resource. On-line at: http://www.javable.com/docs/books/wroxpress/beg_java_webserv/sample/7531-material.pdf (2002)

[Beug99] Beugnard, A., Jezequel, J.-M., Plouzeau, N., Watkins, D.: Making Components Contract Aware. Computer, Vol. 32, No. 7 (July 1999). IEEE-CS (1999) 38-45

[Beus00] Beus-Dukic, L.: Non-Functional Requirements for COTS Software Components. In Proc. of the COTS Wsh.: Continuing Collaborations for Successful COTS Development at ICSE2000 (Limerick, Ireland, June, 2000)

[Bou97] Boutaba, R., El Guemhioui, K. Dini, P.: An Outlook on Intranet Management. IEEE Communications Mag., Vol. 35, No. 10 (Oct. 1997). IEEE (1997) 92-99

[Bro03] Brose, G.: Securing Web Services with SOAP Security Proxies. In Proc. of the 2003 Int. Conf. on Web Services - ICWS’03 (Las Vegas, USA, June 2003). CSREA Press (2003) 231-234

[Cera02] Cerami, E.: Web Services Essentials. 1st edition. O’Reilly & Associates (2002)

[Cerq03] Cerqueira, R., Ansaloni, S., Loques, O.: Deploying Non-Functional Aspects by Contract. In Proc of the The 2nd Wsh. on Reflective and Adaptive Middleware on the Middleware 2003 (Rio de Janeiro, Brazil, June 2003). On-line at: http://www.cs.wustl.edu/~corsaro/papers/RM2003/p1-sztajnberg.pdf (2003)

[Chi03] Chinnici, R., Gudgin, M., Moreau, J.-J., Schlimmer, J., Weerawarana, S. (eds.): Web Services Description Language (WSDL), Ver. 2.0, Part 1: Core Language.

World Wide Web Consortium (W3C) working draft. (Nov. 10, 2003). On-line at: http://www.w3.org/TR/2003/WD-wsdl20-20031110 (2003)

[Chr01] Christensen, E., Curbera, F., Meredith,G., Weerawarana, S.: Web Services Description Language (WSDL) 1.1. World Wide Web Consortium (W3C) note. (Mar. 15, 2001) On-line at: http://www.w3.org/TR/wsdl (2001)

[Cin00] Cingil, I., Dogac, A., Azgin, A.: A broader approach to personalization. Communications of the ACM, Vol. 43, No. 8 (Aug. 2000). ACM (2000) 136-141

[Crn02] Crnkovic, I., Hnich, B., Jonsson, T., Kiziltan, Z.: Specification, Implementation, and Deployment of Components. Communications of the ACM (CACM), Vol. 45, No. 10 (Oct. 2002). ACM (2002) 35-40

[Cur01] Curbera, F., Mukhi, N., Weerawarana, S.: On the Emergence of a Web Services Component Model. In Proc. of the 6th Int. Wsh. on Component-Oriented Programming –WCOP 2001 at the 15th European Conf. on Object-Oriented Programming - ECOOP 2001 (Budapest, Hungary, June 2001). On-line at: http://www.research.microsoft.com/~cszypers/events/WCOP2001/Curbera.pdf (2001)

[Cza03] Czajkowski, K., Dan, A., Rofrano, J., Tuecke, S., Xu, M.: Agreement-based Grid Service Management (OGSI-Agreement), Ver. 0. June 26, 2003. Global Grid Forum. On-line at: http://www.globus.org/research/papers/OGSI_Agreement_2003_06_12.pdf (2003)

[DAM03] The DAML Services Coalition: DAML-S: Semantic Markup for Web Services. WWW resource for DAML-S version 0.9. May 5, 2003. On-line at: http://www.daml.org/services/daml-s/0.9/daml-s.html (2003)

[Dan02] Dan, A., Franck, R., Keller, A., King, R., Ludwig, H.: Web Service Level Agreement (WSLA) Language Specification. In Documentation for the Web Services Toolkit, Ver. 3.2.1. Aug. 9, 2002. IBM (Int. Business Machines) Corporation (2002)

[Dea99] Dean, J., Oberndorf, P., Vigder, M.: Ensuring Successful COTS Development: Wsh. Summary. In Proc. of the First Wsh. on Ensuring Successful COTS Development at ICSE'99 (Los Angeles, USA, May 1999). On-line at: http://wwwsel.iit.nrc.ca/projects/cots/icsewkshp/mainsummary.html (1999)

[Deb03] Debusmann, M., Keller, A.: SLA-driven Management of Distributed Systems using the Common Information Model. In Proc. of the 8th Int. IFIP/IEEE Symp. on Integrated Management – IM 2003 (Colorado Springs, USA, March 2003). Kluwer (2003) 563-576

[Esf04] Esfandiari, B., Tosic, V.: Requirements for Web Service Composition Management. In Proc. of the 11th Hewlett-Packard Open View University Association (HP-OVUA) Wsh. (Paris, France, July 2004). Hewlett-Packard. (2004)

[Far02] Farrell, J. A., Kreger, H: Web Services Management Approaches. IBM Systems Jour., Vol. 41, No. 2. IBM. (2002) 212-227. On-line at: http://www.research.ibm.com/journal/sj/412/farrell.html

[Fer01] Ferguson, D. F.: Web Services Architecture: Direction and Position Paper. In Proc. of the W3C Wsh. on Web Services – WSWS'01 (San Jose, USA, Apr. 2001). W3C. On-line at: http://www.w3c.org/2001/03/WSWS-popa/paper44 (2001)

[Fos02] Foster, I., Keselman, C., Nick, J. M., Tuecke, S.: Grid Services for Distributed Systems Integration. Computer, Vol. 35, No. 6 (June 2002). IEEE-CS (2002) 37-46

[Fro98] Frolund, S., Koistinen, J.: Quality of Service Specification in Distributed Object Systems Design. In Proc. of the 4th USENIX Conf. on Object-Oriented Technologies and Systems - COOTS '98 (Santa Fe, USA, Apr. 1998). USENIX (1998)

[Gou03] Gouscos, D., Kalikakis, M., Georgiadis, P: An Approach to Modeling Web Service QoS and Provision Price. In Proc. of the 1st Int. Web Services Quality Wsh. - WQW 2003 at WISE 2003 (Rome, Italy, Dec. 2003) 1-10

[Hai90] Hailpern, B., Ossher, H.: Extending Objects to Support Multiple Interfaces and Access Control. IEEE Trans. on Software Engineering, Vol. 16, No. 11 (Nov. 1990). IEEE (1990) 1247-1257

[Hau99] Hauck, R., Reiser, H.: Monitoring of Service Level Agreements with Flexible and Extensible Agents. In Proc. of the HP OpenView University Association (HP-OVUA) 6th Plenary Wsh. (Bologna, Italy, Jun 1999). HP. On-line at: http://wwwmnmteam.informatik.uni-muenchen.de/common/Literatur/MNMPub/Publikationen/hare99/HTML-Ver./main.html (1999)

[Hon03] Hondo, M., Kaler, C. (eds.): Web Services Policy Framework (WS-Policy). Ver. 1.01. May 28, 2003. BEA/IBM/Microsoft/SAP. On-line atftp://www6.software.ibm.com/software/developer/library/ws-policy.pdf (2003)

[IBM01] Int. Business Machines Corporation (IBM), Microsoft Corporation.: Web Services Framework. In Proc. of the W3C Wsh. on Web Services – WSWS'01 (San Jose, USA, April 2001). W3C. On-line at: http://www.w3.org/2001/03/WSWS-popa/paper51 (2001)

[Ira01] Irani, R.: An Introduction to ebXML. Web Services Architect Jour.. July 2001.

On-line at: http://www.webservicesarchitect.com/content/articles/irani02.asp

[Jac00] Jacobsen, H.-A., Karamer, B. J.: Modeling Interface Definition Language Extensions. In Proc. of Technology of Object-Oriented Languages and Systems - TOOLS Pacific 2000 (Sydney, Australia, November 2000). IEEE-CS (2000) 241-252

[Kel03] Keller, A., Ludwig, H.: The WSLA Framework: Specifying and Monitoring Service Level Agreements for Web Services. Jour. of Network and Systems Management, Vol. 11, No 1 (Mar. 2003). Plenum Publishing (2003)

[Kep03] Kephart, J.O., Chess, D.M.: The Vision of Autonomic Computing. Computer, Vol. 36, No. 1 (Jan. 2003). IEEE-CS (2003) 41-50

[Kri97] Kristiansen L.: (ed.) Service Architecture, Ver. 5.0. TINA-C (Telecommunications Information Networking Architecture Consortium) specification. (June 16, 1997) On-line: http://www.tinac.com/specifications/documents/sa50-main.pdf (1997)

[Lam03] Lamanna, D.D., Skene, J., Emmerich, W.: SLAng: A Language for Defining Service Level Agreements. In Proc. of the 9th IEEE Wsh. on Future Trends in Distributed Computing Systems - FTDCS 2003 (Puerto Rico, May 2003). IEEE-CS (2003) 100-106

[Lay02] Laymann, F., Roller, D., Schmidt, M.-T.: Web Services and Business Process Management. IBM Systems Jour., Vol. 41, No.2. IBM (2002) 198-211

[Lauk03] Laukkanen, M., Helin, H.: Web Services in Wireless Networks: What Happened to the Performance. In Proc. of the 2003 Int. Conf. on Web Services - ICWS'03 (Las Vegas, USA, June 2003). CSREA Press (2003) 278-284

[LauT01] Lau, T.: QoS for B2B Commerce in the New Web Services Economy. Presented at the Wsh. on Performance and QoS for E-Commerce Applications at ISEC

2001 (Hong Kong, China, Apr. 2001). On-line at: http://www.csd.uwo.ca/research/cords/ISEC2001/lautc.pdf (2001)

[Lew96] Lewis, L.: Implementing Policy in Enterprise Networks. IEEE Communications Mag., Vol. 34, No. 1 (Oct. 1996). IEEE (1996) 50-55

[Lew99] Lewis, L., Ray, P.: Service Level Management Definition, Architecture, and Research Challenges. In Proc. of Global Telecommunications Conf. - Globecom '99 (Rio de Janeiro, Brazil, Dec. 1999). IEEE (1999) 1974-1978

[Lud02] Ludwig, H., Keller, A., Dan, A., King, R.P.: A Service Level Agreement Language for Dynamic Electronic Services. Research Report RC22316 (W0201-112). IBM Research. Jan. 24, 2002. On-line at: http://csdl.computer.org/dl/proceedings/wecwis/2002/1567/00/15670025.pdf (2002)

[Lud03] Ludwig, H., Keller, A., Dan, A., King, R.P., Franck, R.: Web Service Level Agreement (WSLA) Language Specification, Ver. 1.0, Revision wsla-2003/01/28. Int. Business Machines Corporation (IBM). On-line at: http://www.research.ibm.com/wsla/WSLASpecV1-20030128.pdf (2003)

[Lup97] Lupu, E., Sloman, M.: A Policy Based Role Object Model. In Proc. of the 1st Int. Enterprise Distributed Object Computing Conf. – EDOC ’97 (Gold Coast, Australia, Oct. 1997). IEEE-CS (1997)

[Lup99] Lupu, E., Sloman, M.: Conflicts in Policy-Based Distributed Systems Management. IEEE Trans. on Software Engineering (Special Issue on Inconsistency Management), Vol. 25, No. 6 (Nov./Dec. 1999). IEEE (1999) 852-869

[Maa04] Maamar, Z., Sheng, Q.Z., Benatallah, B. (2004) On Composite Web Services Provisioning in an Environment of Fixed and Mobile Computing Resources. Informa-

tion Technology and Management, Vol. 5, No. 3. Kluwer Academic Publishers (2004) 251-270

[Mac02] Machiraju, V., Sahai, A., van Moorsel, A.: Web Services Management Network: An Overlay Network for Federated Service Management. Research Report HPL-2002-234. Hewlett-Packard (HP) Laboratories Palo Alto. Aug. 21, 2002. Online at: http://www.hpl.hp.com/techreports/2001/HPL-2002-234.pdf A shorter version in Proc. of the Eight Int. Symp. on Integrated Network Management - IM 2003 (Colorado Springs, USA, Mar. 2003). IEEE (2003) 351-364

[Marc03] Marchetti, C., Pernici, B., Plebani, P.: A Quality Model for e-Service based Multi-Channel Adaptive Information Systems. In Proc. of the 1st Int. Web Services Quality Wsh. - WQW 2003 at WISE 2003 (Rome, Italy, Dec. 2003) 47-54

[Marc04] Marchetti, C., Pernici, B., Plebani, P.: A Quality Model for Multichannel Adaptive Information Systems. In Proc. of the 13th Int. World Wide Web Conf. - WWW 2004 (New York, USA, May 2004). ACM (2004) 48-54

[Mart03] Martin-Diaz, O., Ruiz-Cortes, A., Corchuelo, R., Toro, M.: A Framework for Classifying and Comparing Web Services Procurement Platforms. In Proc. of the 1st Int. Web Services Quality Wsh. - WQW 2003 at WISE 2003 (Rome, Italy, Dec. 2003) 37-46

[MaW04] Ma, W.: Towards Web Service Management – A Prototype of Web Service Offerings Infrastructure (WSOI). M.A.Sc. thesis, Carleton University, Ottawa, Canada. To be published. (2004)

[McG03] McGregor, C.: A Method to Extend BPEL4WS to Enable Business Performance Measurement. In Proc. of the 2003 Int. Conf. on Web Services - ICWS'03 (Las Vegas, USA, June 2003). CSREA Press (2003) 46-51

[Mck99] Mckee, P., Marshall, I.: Behavioural Specification using XML. In Proc. of the 7th IEEE Wsh. on Future Trends of Distributed Computing Systems - FTDCS'99, (Cape Town, South Africa, Dec. 1999). IEEE-CS (1999) 53-59

[Men01] Mennie, D., Pagurek, B.: A Runtime Composite Service Creation and Deployment Infrastructure and Its Applications in Internet Security, E-commerce, and Software Provisioning. In Proc. of the 25th Ann. Int. Computer Software and Applications Conf. - COMPSAC 2001 (Chicago, USA, Oct. 2001). IEEE-CS (2001) 371-376

[Mey92] Meyer, B.: Applying "Design by Contract". Computer, Vol. 25, No. 10 (Oct. 1992). IEEE-CS (1992) 40-51

[Mil03] Milenkovic, M., Robinson, S. H., Knauerhase, R. C., Barkai, D., Garg, S., Tewari, V., Anderson, T. A., Bowman, M.: Toward Internet Distributed Computing. Computer, Vol. 36, No. 5 (May 2003). IEEE-CS (2003) 38-46

[Nea03] Neal, S., Cole, J., Linington, P. F., Milosevic, Z., Gibson, S. Kulkarni, S.: Identifying requirements for Business Contract Language: a Monitoring Perspective. In Proc. of the Seventh Int. Enterprise Distributed Object Computing Conf. – EDOC'03 (Brisbane, Australia, Sep. 2003). IEEE-CS (2003) 50-61

[OAS02] OASIS ebXML Collaboration Protocol Profile and Agreement Technical Committee: Collaboration-Protocol Profile and Agreement Specification, Ver. 2.0. Sep. 23, 2002. OASIS (Organization for the Advancement of Structural Information Standards). On-line at: http://www.ebxml.org/specs/ebcpp-2.0.pdf (2004)

[OAS04] OASIS: ebXML – Enabling A Global Electronic Market. WWW resource. OASIS (Organization for the Advancement of Structural Information Standards). On-line at: http://www.ebxml.org/ (2004)

[OMG03] Object Management Group (OMG): OCL Language Description. Chapter 7 in UML 2.0 OCL Specification. OMG Adopted Specification. Oct. 2003. OMG. On-line at: ftp://ftp.omg.org/pub/docs/ptc/03-10-14.pdf (2003) 5-32

[Ore98] Oreizy, P., Medvidovic, N., Taylor, R. N.: Architecture-Based Software Runtime Evolution. In Proc. of the Int. Conf. on Software Engineering 1998 - ICSE'98 (Kyoto, Japan, Apr. 1998). ACM (1998) 177-186

[Ouy03] Ouyang, J., Chu, W., Savur, P.: WTM: A Traffic Monitoring Framework for Web Service Trans. In Proc. of WWW203 – The Twelfth Int. World Wide Web Conf. (Budapest, Hungary, May 2003). ACM. On-line at: http://www2003.org.cdrom/papers/alternate/P632/p632-ouyang.htm (2003)

[Pat03a] Patel, K.: XML Grammar and Parser for the Web Service Offerings Language. M.A.Sc. thesis, Carleton University, Ottawa, Canada. Jan. 30, 2003. On-line at: http://www.sce.carleton.ca/netmanage/papers/KrutiPatelThesisFinal.pdf (2003)

[Pat03b] Patel, K., Pagurek, B., Tosic, V.: Improvements in WSOL Grammar and "Premier" WSOL Parser. Res. Rep. SCE-03-25, Department of Systems and Computer Engineering, Carleton University, Ottawa, Canada, Oct. 2003. On-line at: http://www.sce.carleton.ca/netmanage/papers/PatelEtAlResRepOct2003.pdf (2003)

[Pel03] Peltz, C.: Web Services Orchestration and Choreography. Computer, Vol. 36, No. 10 (Oct. 2003) IEEE-CS (2003) 46-52

[Pot03] Potts, M., Sedukhin, I., Kreger, H., Stokes, E.: Web Service Manageability - Specification (WS-Manageability). Ver. 1.0. Sept. 2003. Submitted to the OASIS Web Services Distributed Management Technical Committee. IBM/CA/Talking Blocks. On-line at: http://www.oasis-open.org/committees/download.php/3471/Web%20Service%20Manageability%20-%20Specification%201.0.pdf (2003)

[Pry99] Pryce, N., Dulay, N.: Dynamic Architectures and Architectural Styles for Distributed Programs. In Proc. of 7th IEEE Wsh. on Future Trends of Distributed Computing Systems - FTDCS'99 (Cape Town, South Africa, December 1999), IEEE Computer Society Press, 89 –94

[Raj99] Rajan, R., Verma, D., Kamat, S., Felstaine, E., Herzog, S.: A Policy Framework for Integrated and Differentiated Services in the Internet. IEEE Network, Vol. 13, No. 5 (Sep. 1999). IEEE (1999) 36-41

[Ram98] Raman, L.: OSI Systems and Network Management. IEEE Communications Mag., Vol. 36, No. 3 (March 1998). IEEE (1998) 46-53

[Sah02a] Sahai, A., Durante, A., Machiraju, V.: Towards Automated SLA Management for Web Services. Research Report HPL-2001-310 (R.1). Hewlett-Packard (HP) Laboratories Palo Alto. July 26, 2002. On-line at: http://www.hpl.hp.com/techreports/2001/HPL-2001-310R1.pdf (2002)

[Sah02b] Sahai, A., Machiraju, V., Sayal, M., van Moorsel, A., Casati, F.: Automated SLA Monitoring for Web Services. In Proc. of the 13th IFIP/IEEE Int. Wsh. on Distributed Systems: Operations and Management, DSOM 2002 (Montreal, Canada, Oct. 2002). Lecture Notes in Computer Science (LNCS), No. 2506. Springer-Verlag (2002) 28-41

[Sal04] Salle, M., Bartolini, C.: Management by Contract. In Proc. of the 2004 IFIP/IEEE Int. Symp. on Network Operations and Management - NOMS 2004 (Seoul, South Korea, Apr. 2004). IEEE (2004) 787- 800

[Sch02] Schlimmer, J.C. (ed.): Web Services Description Requirements. World Wide Web Consortium (W3C) working draft. Oct. 28, 2002. On-line at: http://www.w3.org/TR/2002/WD-ws-desc-reqs-20021028/ (2002)

[Shai03] ShaikhAli, A., Rana, O. F., Al-Ali, R., Walker, D. W.: UDDIe: An Extended Registry for Web Services. In Proc. of 2003 Symp. on Applications and the Internet (SAINT'03) Wsh.s, Wsh. on Services Oriented Computing: Models, Architectures and Applications (Orlando, USA, Jan. 2003). IEEE-CS (2003) 85-89

[Shar03] Sharma, A., Adarkar, H., Sengupta, S.: Managing QoS through Prioritization in Web Services. In Proc. of the 1st Int. Web Services Quality Wsh. - WQW 2003 at the 4th Int. Conf. on Web Information Systems Engineering - WISE 2003 (Dec. 2003, Rome, Italy) 20-28

[Slo94] Sloman, M.: Policy Driven Management for Distributed Systems. Jour. of Network and Systems Management, Vol. 2, No. 4 (December 1994), Plenum Press, 333-360

[Slo95] Sloman, M.: Management Issues for Distributed Services. In Proc. of the IEEE Second Int. Wsh. on Services in Distributed and Networked Environments – SDNE '95 (Whistler, Canada, June 1995). IEEE-CS Press (1995) 52-59

[Sne02] Snell, J., Tidwell, D., Kulchenko, P.: Programming Web Services with SOAP. 1st edition. O'Reilly & Associates (2002)

[Syc03] Sycara, K.: Autonomous Semantic Web Services. Invited talk at CAiSE'03

(Velden, Austria, June 2003)

[Tec02] TechMetrics Research: Axis: the New Incarnation of Apache SOAP. White paper. TechMetrics Research (2002)

[Tho01] Thomas Manes, A.: Enabling Open, Interoperable, and Smart Web Services – The Need for Shared Context. In Proc. of the W3C Wsh. on Web Services – WSWS'01 (San Jose, USA, April 2001). On-line at: http://www.w3.org/2001/03/wsws-popa/paper29 (2001)

[Tia03] Tian, M.,Gramm, A., Naumowicz, T., Ritter, H., Jchiller, J.: A Concept for QoS Integration in Web Services. In Proc. of the 1st Int. Web Services Quality Wsh. - WQW 2003 at the 4th Int. Conf. on Web Information Systems Engineering - WISE 2003 (Rome, Italy, Dec. 2003) 29-36

[Tos98] Tosic, V.: On Object-Oriented Information Specification in Network and System Management. M.Sc. thesis. Faculty of Electronic Engineering, University of Nis, Yugoslavia (Nov. 1998)

[Tos01] Tosic, V., Mennie, D., Pagurek, B.: Software Configuration Management Related to Management of Distributed Systems and Services and Advanced Service Creation. In Proc. of the Tenth Int. Wsh. on Software Configuration Management (SCM-10) at ICSE 2001 (Toronto, Canada, May 2001). ACM. (2001) Extended version "Software Configuration Management Related to the Management of Distributed Systems and Service Oriented Architectures" in Westfechtel, B., van der Hoek, A. (Eds.) Software Configuration Management. Lecture Notes in Computer Science (LNCS), No. 2649. Springer-Verlag (2003) 54-69

[Tos02a] Tosic, V., Pagurek, B., Esfandiari, B., Patel, K.: On Various Approaches to Dynamic Adaptation of Distributed Component Compositions. Res. Rep. OCIECE-02-02. Ottawa-Carleton Institute for Electrical and Computer Engineering (OCIECE). June 2002. On-line at: http://www.sce.carleton.ca/netmanage/papers/TosicEtAlResRepJune2002.pdf (2002)

[Tos02b] Tosic, V., Esfandiari, B., Pagurek, B., Patel, K.: On Requirements for Ontologies in Management of Web Services. In Proc. of the Wsh. on Web Services, e-Business, and the Semantic Web at CAiSE'02 (Toronto, Canada, May 2002). Lecture Notes in Computer Science (LNCS), No. 2512. Springer-Verlag (2002) 237-247

[Tos02d] Tosic, V., Pagurek, B., Esfandiari, B., Patel, K., Ma, W.: Web Service Offerings Language (WSOL) and Web Service Composition Management (WSCM). In Proc. of the OOWS'02 (Object-Oriented Web Services) Wsh. at OOPSLA 2002 (Seattle, USA, Nov. 2002) On-line at: http://www.research.ibm.com/people/b/bth/OOWS2002/tosic.zip (2002)

[Tos02d] Tosic, V., Pagurek, B., Patel, K.: WSOL – A Language for the Formal Specification of Various Constraints and Classes of Service for Web Services. Res. Rep. OCIECE-02-06. Ottawa-Carleton Institute for Electrical and Computer Engineering. Nov. 15, 2002. On-line at: http://www.sce.carleton.ca/netmanage/papers/TosicEtAlResRepNov2002.pdf (2002)

[Tos03a] Tosic, V., Patel. K., Pagurek, B.: Reusability Constructs in the Web Service Offerings Language (WSOL). In Proc. of the Wsh. on Web Services, e-Business, and the Semantic Web (WES) at CAiSE'03 (Velden, Austria, June 2003). Lecture Notes in Computer Science (LNCS). Springer-Verlag. Extended version: Res. Rep. SCE-03-

21, Department of Systems and Computer Engineering, Carleton University, Ottawa, Canada, Sep. 2003. On-line at: http://www.sce.carleton.ca/netmanage/papers/TosicEtAlRepSeptember2003.pdf (2003)

[Tos03b] Tosic, V., Patel, K., Pagurek, B.: WSOL – A Language for the Formal Specification of Classes of Service for Web Services. Proc. of ICWS'03 (Las Vegas, USA, June 2003). CSREA Press (2003) 375-381

[Tos03c] Tosic, V., Ma, W., Pagurek, B., Esfandiari, B.: On the Dynamic Manipulation of Classes of Service for XML Web Services. In Proc. of the 10th Hewlett-Packard Open View University Association (HP-OVUA) Wsh. (Geneva, Switzerland, July 2003). Hewlett-Packard. Extended version: Research Report SCE-03-15, Department of Systems and Computer Engineering, Carleton University, Ottawa, Canada, June 2003. On-line at: http://www.sce.carleton.ca/netmanage/papers/TosicEtAlResRepJune2003.pdf (2003)

[Tos04a] Tosic, V., Ma, W., Pagurek, B., Esfandiari, B.: Web Service Offerings Infrastructure (WSOI) – A Management Infrastructure for XML Web Services. In Proc. of NOMS (IEEE/IFIP Network Operations and Management Symp.) 2004, Seoul, South Korea, April 19-23, 2004. IEEE (2004) 817-830

[Tos04b] Tosic, V., Pagurek, B., Patel, B. Esfandiari, B., Ma, W.: Management Applications of the Web Service Offerings Language (WSOL). To be publ. in Information Systems. Elsevier. Early version: Proc. of CAiSE'03 (Velden, Austria, June 2003). Lecture Notes in Computer Science (LNCS), No. 2681. Springer-Verlag (2003) 468-484

[Tos04c] Tosic, V.: Overview of Some Industrial Products for Web Service Management (WSM) and Web Service Composition Management (WSCM). Res. Rep. SCE-04-06, Department of Systems and Computer Engineering, Carleton University, Ottawa, Canada. (May 2004)

[Tos04d] Tosic, V.: WSOL Version 1.2. Res. Rep. SCE-04-11, Department of Systems and Computer Engineering, Carleton University, Ottawa, Canada. (July 2004)

[Tos04e] Tosic, V.: On the Algorithms and Protocols for Manipulation of Service Offerings. Res. Rep. SCE-04-12, Department of Systems and Computer Engineering, Carleton University, Ottawa, Canada. (Aug. 2004)

[Tov04] Toval, A., Requena, R., Luis Fernández, J: OCL Tools. WWW resource. Apr. 7, 2004. On-line at: http://www.um.es/giisw/ocltools/ (2004)

[vMo02] van Moorsel, A.: Ten-Step Survival Guide for the Emerging Business Web. In Proc. of the Wsh. on Web Services, e-Business, and the Semantic Web at CAiSE'02 (Toronto, Canada, May 2002). Lecture Notes in Computer Science (LNCS), No. 2512. Springer-Verlag (2002) 1-11

[W3C01] World Wide Web Consortium (W3C). W3C Wsh. on Web Services. Proceedings (San Jose, USA, April 2001), W3C. On line at: http://www.w3.org/2001/01/WSWS

[Web02] WebServices.org: Introduction to Axis. WWW resource. May 28, 2002. WebServices.org. On-line at: http://www.webservices.org/index.php/article/articleview/415/1/24/ (2002)

[XuL03] Xu, L., Jeusfeld, M.A.: Pro-active Monitoring of Electronic Contracts. In Proc. of CAiSE'03 (Velden, Austria, June 2003). Lecture Notes in Computer Science (LNCS), No. 2681. Springer-Verlag (2003) 584 – 600

[Zin97] Zinky, J.A., Bakken, D.E., Schantz, R.E.: Architectural Support for Quality of Service for CORBA Objects. Theory and Practice of Object Systems, Vol. 3, No. 1. John Wiley & Sons (Apr. 1997)

Appendix A. Sub-protocols for Consumer-initiated Switching

Notes for all sub-protocols:

- I have assumed that consumer *C* is never the same entity as provider *P*.
- I have assumed that all communication between management parties is in the form request-reply.
- I have not assumed that communication between management parties is synchronous. In the case of synchronous communication, processing of previously submitted requests has to be completed before a new request is submitted.
- I have assumed that there are no cyclic dependencies in finalization activities performed by various management parties. In the case of a such cyclic dependency, the results of incomplete finalization activities at a management party X are used for finalization activities at a management party Y, which are in turn (directly or indirectly) used to complete finalization activities at the management party X.
- In all SOAP messages, session ID is transported between the participating management parties.
- Additional information, such as UML sequence diagrams, for these sub-protocols can be found in [Tos04e].

A.1 Switching Beginning (Initiation) – Sub-protocol 'B'

Participants: In the most general case, participants are a provider *P*, a consumer *C*, an independent accounting party *AP1*, and all independent management third parties in *MP1*. See Table 4.1 for definitions.

Before: Provider *P* has at least two service offerings *SO1* and *SO2*. *SO1* and *SO2* differ in management third parties, including independent accounting parties. Consumer *C* uses *SO1*. *C* knows what service offerings from *P* it can use and what of these service offerings are currently active. The consumer gets this information when it invokes the operation '*listSOsForMe()*', e.g., at the beginning of the session. Whenever the provider activates, deactivates, creates a service offering, or changes access rights for an existing service offering, it should notify consumers that can use it. The consumer can also invoke the '*listActiveSOsForMe()*' operation to get an up-to-date list of active service offerings it can use.

Trigger: *C* determines that it would be better for it to use *SO2* instead of *SO1*.

Purpose: To prepare all management parties participating in the old service offering *SO1* for switching. An important part of this preparation is to finish processing of already submitted consumer requests using *SO1* and to block and queue processing of any new consumer requests until the end of switching. Provider-side checks determining whether the switching is possible are also performed in this sub-protocol.

After: If *C* has the right to use *SO2* and *SO2* is currently active, then all management parties participating in *SO1* are prepared for switching. All requests from *C* submitted before the switching request are processed using *SO1*. The processing of all requests from *C* submitted after the switching request are blocked and queued at *AP1*.

Sub-protocol steps:

1. The consumer *C* sends a SOAP message that invokes the '*switchSO()*' operation of the accounting party *AP1* used in *SO1*.

2. Using the internal '*block()*' operation, *AP1* blocks and queues any further requests from *C* for operations outside the Service Offering Management (SOM) ports. All consumer requests that were submitted before the switching request are not blocked by *AP1* and are processed using *SO1*. Parts of the latter processing can occur in parallel with the rest of this sub-protocol.
3. *AP1* forwards to the provider *P* the SOAP message invoking the '*switchSO()*' operation, along with the information about the last request from *C* that was not blocked.
4. After *P* finishes processing, using *SO1*, of all requests that were not blocked by *AP1*, it checks whether it can switch *C* to *SO2* or not. The internal '*checkSwitch()*' operation is used for this purpose. Two cases are possible:
    a. *P* cannot switch *C* to *SO2*. In this case, *P* replies, 'False' to *AP1*. Then, in the next step *AP1* replies 'False' to *C* and unblocks all queued requests, so they are processed with *SO1*. *C* continues to use *SO1*.
    b. *P* can switch *C* to *SO2*, so the switching between service offerings is accepted and will be performed. I assume this case in the following sub-protocol steps.
5. *P* sends a SOAP message to *AP1* as a reply to its '*switchSO()*' invocation. Since I assumed that switching is accepted, *P* replies, 'True' to *AP1*.
6. After *AP1* finishes processing, using *SO1*, of responses to all consumer requests that it did not block, it sends a SOAP message to *C* as a reply to its '*switchSO()*' invocation. Since I assumed that switching is accepted, *AP1* replies, 'True' to *C*.

<u>Analysis of the number of exchanged SOAP messages:</u> In this sub-protocol, at most 4 SOAP messages are exchanged – 2 requests and 2 corresponding replies. However, if the

provider *P* plays the role of the accounting party for *SO1* and no independent *AP1* exists, then only 2 SOAP messages are exchanged – 1 request from *C* to *P* and 1 corresponding reply. Therefore, the number of exchanged SOAP messages in this sub-protocol, using the notation from Table 4.4, is: $2*(A_1+1)$.

Possibilities for optimizations: If this sub-protocol is used with other sub-protocols of for consumer-initiated switching of service offerings, then Step 5 and Step 6 can be omitted/postponed. These two steps represent response messages that can be combined together with later SOAP messages flowing in the same direction.

Reusability: This sub-protocol is not reusable as a whole for provider-initiated switching. The internal operations '*block()*' and '*checkSwitch()*' are also used for provider-initiated switching, but there is a difference in messages exchanged between the participating management parties.

A.2 Initialization of Parties for the New Service Offering – Sub-protocol 'I'

Participants: In the most general case, the participants are a provider *P*, a consumer *C*, an independent accounting party *AP2*, and all independent management third parties in *MP2*. See Table 4.1 for definitions.

Before: The sub-protocol 'B' for switching beginning (initiation) was executed and the switching was accepted.

Trigger: The end of sub-protocol 'B'. (Actually, this sub-protocol could start as soon as the Step 4 of sub-protocol 'B' is completed, because it can partially execute in parallel with the Steps 5 and 6 of sub-protocol 'B'.)

Purpose: To initialize all management parties participating in monitoring activities for the new service offering *SO2*. Examples of initialization activities are making an association between the current session ID and the service offering name, as well as creation and/or initialization of appropriate data structures. I denote this sub-protocol as 'I' (for 'initialization').

After: If there are no unexpected errors, all participating parties have performed necessary initialization activities for the new service offering *SO2* and they are ready to be used.

Sub-protocol steps:

1. The provider *P* invokes its internal operation '*initializeWork()*'.
2. The provider *P* sends a SOAP message to the accounting party *AP2* to invoke the '*initializeWork()*' operation of *AP2*.
3. Before and/or after the accounting party *AP2* performs its initialization activities, it invokes the '*initializeWork()*' operation of all *MP2* management third parties used for *SO2*. If there is more than one management third party in *MP2*, these invocations can be performed sequentially or in parallel. For every such invocation, one SOAP message is sent.
4. After a management third party in *MP2* performs its initialization activities, it sends a reply SOAP message to the accounting party *AP2*. There are as many reply SOAP messages as there are management third parties in *MP2*.
5. When *AP2* receives responses from all management third parties in *MP2* and completes its internal initialization processes and there is no error, it sends a response SOAP message to the provider *P* about the success of the initialization processes.

6. Independently from Steps 1 to 5 of this sub-protocol, but after the completion of Step 6 from the sub-protocol 'B', the consumer *C* invokes its internal operation '*initializeWork()*'.

Possible exceptions: If the '*initializeWork()*' operation for any management party cannot successfully perform initialization activities (even after some retries), then the switching to *SO2* is not possible. It is cancelled and the rollback process is started. In such a case, both the provider and the consumer are informed and all previous initialization activities related to *SO2* are reversed. Depending on the reason why initialization is not possible, *SO2* might have to be deactivated. *AP1* unblocks all queued requests, so they are processed with *SO1*. *C* continues to use *SO1*.

Analysis of the number of exchanged SOAP messages: In this sub-protocol, the number of exchanged SOAP messages depends on the number of management third parties in *MP2* and whether the used accounting party is an independent party or not. Using the notation from Table 4.4, the number of exchanged SOAP messages is: $2*(A_2+M_2)$.

Possibilities for optimizations: If this sub-protocol is used with other sub-protocols of for consumer-initiated switching of service offerings, then it might be possible to combine initialization of management parties for the new service offering *SO2* and finalization of management parties for the old service offering *SO1*. In such cases, one operation '*initAndFinal()*' is used instead of a pair of operations '*initializeWork()*' and '*initializeWork()*'. Further optimization can be achieved when all SOAP messages from *AP2* to management parties in *MP2* are sent in parallel, e.g., using multicast. Another possibility for optimization is not to use any response SOAP message in this sub-protocol. Then, if

no fault SOAP message is sent back, the requesting party assumes that initialization completed without problems.

Reusability: This sub-protocol is also reusable for provider-initiated switching.

A.3 Finalization of Parties for the Old Service Offering – Sub-protocol 'F'

Participants: In the most general case, the participants are a provider *P*, a consumer *C*, an independent accounting party *AP1*, and all independent management third parties in *MP1*. See Table 4.1 for definitions.

Before: The sub-protocol 'B' for switching beginning (initiation) was executed and the switching was accepted. Then, the sub-protocol 'I' for initialization of parties for the new service offering was executed and no error was reported.

Trigger: The end of sub-protocol 'I'.

Purpose: To finalize all *SO1*-related activities in all management parties participating in monitoring activities for the old service offering *SO1* and to collect all information about *SO1* in the accounting party *AP1*. Examples of finalization activities are calculation of total prices and penalties to be paid and deletion of *SO1*-related data structures or their parts that are no longer needed.

After: If there are no unexpected errors, all participating parties have performed necessary finalization activities for the old service offering *SO1* and *AP1* has calculated total prices and penalties to be paid.

Sub-protocol steps:

1. The provider *P* invokes its internal operation '*finalizeWork()*'.

2. After performing finalization activities related to the old service offering *SO1*, the provider *P* sends a SOAP message to the accounting party *AP1* to invoke the '*finalizeWork()*' operation of *AP1*. This reply message also contains all *SO1*-related management information that *P* has not yet sent *AP1*.
3. The accounting party *AP1* invokes the '*finalizeWork()*' operation of all *MP1* management third parties used for *SO1*. If there is more than one management third party in *MP1*, these invocations can be performed sequentially or in parallel. For every such invocation, one SOAP message is sent.
4. After a management third party in *MP1* performs its finalization activities, it sends a reply SOAP message to the accounting party *AP1*. This reply message contains all *SO1*-related management information that this management third party has not yet sent *AP1*. There are as many reply SOAP messages as there are management third parties in *MP1*.
5. The accounting party *AP1* sends a SOAP message to the consumer *C* to invoke its operation '*finalizeWork()*'.
6. After performing finalization activities related to the old service offering *SO1*, the consumer *C* sends a reply SOAP message back to *AP1*. This reply message contains all *SO1*-related management information that *C* has not yet sent *AP1*.
7. After the accounting party *AP1* performs its own finalization activities, it sends a reply SOAP message to the provider *P*. This reply message might contain total *SO1*-related prices and penalties to be paid and/or other *SO1*-related management information that *P* might want to know.

Additional assumptions: Note that the above sub-protocol assumes that finalization activities at the provider side can be performed without additional information obtained in finalization of the other management parties. If this is not the case, then *AP1* has to additionally invoke the '*finalizeWork()*' operation of the provider after invoking this operation of the other management parties. This request and the subsequent response require a SOAP message each. On the contrary, the above sub-protocol makes no such assumption for the finalization activities performed at the consumer side. Also the actual order of sending '*finalizeWork()*' requests to various management parties can differ from the above sub-protocol, but this does not increase the number of exchanged SOAP messages. For example, it might be needed to invoke '*finalizeWork()*' of the consumer before invoking the same operation of some management party which uses the information produced by the consumer.

Possible exceptions: If the '*finalizeWork()*' operation for any management party cannot successfully perform finalization activities (even after some retries), then the switching to *SO2* is still possible. In such a case, the accounting party *AP1* proceeds without the data from the management party that returned the exception (i.e., the fault SOAP message). *AP1* has to inform both the provider *P* and the consumer *C* about such exceptions.

Analysis of the number of exchanged SOAP messages: In this sub-protocol, the number of exchanged SOAP messages depends on the number of management third parties in *MP1* and whether the consumer and/or the provider perform evaluation of constraints or not. Using the notation from Table 4.4, the number of exchanged SOAP messages is: $2*(A_1+M_1+C_1)$. If *AP1* has to invoke the '*finalizeWork()*' operation of *P*, then the number of exchanged SOAP messages is: $2*(A_1+M_1+C_1+P_1)$.

Possibilities for optimizations: This sub-protocol can be optimized in several ways. First, the if this sub-protocol is executed as a part of consumer-initiated switching, then there is no need to perform Steps 2 and 7 of this sub-protocol. Instead, Step 5 from the sub-protocol 'B' for switching beginning (initiation) can be postponed and merged with Step 2 of this sub-protocol. The saving is: A_1 SOAP messages. Then, Step 7 can be postponed and executed as part of the sub-protocol 'N' for notification of the consumer and the provider about the completed switching. Further, if this sub-protocol is used with other sub-protocols for consumer-initiated switching of service offerings, then it is possible to combine initialization of management parties for the new service offering *SO2* and finalization of management parties for the old service offering *SO1*. In such cases, one operation '*initAndFinal()*' is used instead of a pair of operations '*initializeWork()*' and '*finalizeWork()*'. Further optimization can be achieved when all SOAP messages from *AP1* to management parties in *MP1* are sent in parallel, e.g., using multicast.

Reusability: This sub-protocol is also reusable for provider-initiated switching.

A.4 Relaying (Forwarding) of Requests to the New Accounting Party – Sub-protocol 'R'

Participants: In the most general case, the participants are a consumer *C*, an independent accounting party *AP1*, and an independent accounting party *AP2*. See Table 4.1 for definitions.

Before: The sub-protocol 'B' for switching beginning (initiation) was executed and the switching was accepted. Then, the sub-protocol 'I' for initialization of parties for the new

service offering was executed and no error was reported. Afterwards, the sub-protocol ‘F’ for finalization of parties for the old service offering was executed.

Trigger: The end of sub-protocol ‘F’.

Purpose: To transfer information about all consumer requests submitted after the switching initiation. In the sub-protocol ‘B’, *AP1* blocked and queued processing of consumer requests submitted after the switching initiation. These requests are to be processed with *SO2*, so they have to be transferred to *AP2*.

After: If there are no unexpected errors, all consumer requests blocked and queued at *AP1* are now transferred to *AP2*. They will be processed *SO2*.

Sub-protocol steps:

1. The old accounting party *AP1* sends a SOAP message to the new accounting party *AP2* to invoke its operation ‘*forwardRequests()*’.
2. The new accounting party *AP2* sends a reply SOAP message to the old accounting party *AP1*. This SOAP message is only a confirmation of the receipt of ‘*forwardRequests()*’. *AP2* continues to process the forwarded requests using *SO2*. These requests are later forwarded to management third parties from *MP2* and the provider *P*. The results and possible exceptions are passed directly to the consumer *C*, without passing through *AP1*.

Possible exceptions: If the ‘*forwardRequests()*’ operation of *AP2* cannot be completed successfully, then *C* will have to resubmit these requests directly to *AP2*.

Analysis of the number of exchanged SOAP messages: In this sub-protocol, the number of exchanged SOAP messages depends on whether the accounting parties in the old

and the new service offering are the same. Using the notation from Table 4.4, the number of exchanged SOAP messages is: $2*(D_{1:2})$.

Possibilities for optimizations: Currently, I do not see possibilities for optimization of this sub-protocol.

Reusability: This sub-protocol is also reusable for provider-initiated switching.

A.5 Notification of the Consumer and the Provider about the Completed Switching – Sub-protocol 'N'

Participants: In the most general case, the participants are a consumer *C*, a provider *P*, and an independent accounting party *AP1*. See Table 4.1 for definitions.

Before: The sub-protocol 'B' for switching beginning (initiation) was executed and the switching was accepted. Then, the sub-protocol 'I' for initialization of parties for the new service offering was executed and no error was reported. Afterwards, the sub-protocol 'F' for finalization of parties for the old service offering was executed and no error was reported. Finally, the sub-protocol 'R' for relaying (forwarding) of requests to the new accounting party was executed.

Trigger: The end of sub-protocol 'R'.

Purpose: To inform the consumer and the provider that the switching has been successfully completed.

After: All requests from the consumer *C* to the provider *P* are processed using *SO2*.

Sub-protocol steps:

1. The old accounting party *AP1* sends a SOAP message to the provider *P* to inform it that switching *C* from *SO1* to *SO2* was completed successfully. This is achieved with the '*switichingComplete()*' operation.
2. The provider *P* sends a reply/confirmation SOAP message to *AP1*.
3. The old accounting party *AP1* sends a SOAP message to the consumer *C* to inform it that switching from *SO1* to *SO2* of *P* was completed successfully. This is achieved with the '*switichingComplete()*' operation.
4. The consumer *C* sends a reply/confirmation SOAP message to *AP1*.

Possible exceptions: If a notification of the consumer and/or the provider produces an error, this step has to be retried. Retries are repeated until the maximum number of retries or a timeout is reached. If it is still not possible to notify the consumer and/or provider, the session between them has to be ended because they are no longer available.

Analysis of the number of exchanged SOAP messages: In this sub-protocol, the number of exchanged SOAP messages depends on whether the accounting party *AP1* in the old service offering is independent. Using the notation from Table 4.4, the number of exchanged SOAP messages is: $2*(A_1+1)$.

Possibilities for optimizations: If this sub-protocol is executed as a part of consumer-initiated switching, then there are significant opportunities for optimization. Step 1 of this sub-protocol can be combined together with Step 7 of the sub-protocol 'F' for finalization of parties for the old service offering. On the other hand, Step 3 of this sub-protocol can be combined together with Step 6 of the sub-protocol 'B' for switching beginning (initiation). Then Step 4 of this sub-protocol becomes obsolete. The total savings are: A_1+2*1 SOAP messages.

Reusability: This sub-protocol is also reusable for provider-initiated switching.

A.6 Total Number of Exchanged SOAP Messages in Consumer-initiated Switching

The total number of number of exchanged SOAP messages in consumer-initiated switching is determined as a sum of the numbers of exchanged messages in all sub-protocols minus the number of exchanged messages saved in optimizations.

The sum of the numbers of exchanged messages in all sub-protocols is:

$CSOM = B+I+F+R+N =$

$2*(A_1+1)+2*(A_2+M_2)+2*(A_1+M_1+C_1)+2*(D_{1:2})+2*(A_1+1) =$

$2*(2+3*A_1+M_1+C_1+A_2+M_2+D_{1:2})$

A.7 Possible Optimizations

The following combinations of SOAP messages are simple to perform and result in optimizations:

- Step 5 of sub-protocol 'B' and Step 2 of sub-protocol 'F',
- Step 7 of sub-protocol 'F' and Step 1 of sub-protocol 'N', and
- Step 6 of sub-protocol 'B' and Step 3 of sub-protocol 'N'.

A consequence of the latter combination is that Step 4 from sub-protocol 'N' becomes obsolete. Savings for these 4 mentioned optimizations are:

$2*(A_1+1)$ SOAP messages.

These optimizations disturb modularization, so I will use the term 'not modularized' to refer to protocol variants that use them.

When sub-protocols 'I' and 'F' are performed together, the savings are:

$2*(S_{1:2}+E_{1:2})$ SOAP messages.

In a sub-case of the above case, *AP1* initializes and finalizes all management parties in *MP1* and *MP2*, as well as *AP2*. Initialization/finalization of *AP2* and relaying of messages from *AP1* to *AP* can be performed simultaneously, so additional savings are:

$2*(A_2)$ SOAP messages.

When *AP1* is equal to *AP2*, then this accounting party can avoid explicitly informing the provider about the results of finalization and switching. The provider can assume that finalization and switching were successful, and the particular finalization results can be piggybacked into a future regular operation request from the consumer. The savings are:

$2*(A_1)$ SOAP messages.

AP1 can simply reject all consumer requests during the switching process. In this case, sub-protocol ‘R’ is not needed and savings are:

$2*(D_{1:2})$ SOAP messages,

but only if the previously mentioned optimization $2*(A_2)$ is not applied.

Some savings are possible if operations are not request-reply.

It might be also possible that *AP1* sends in parallel messages to independent management parties in *MP1*, while *AP2* does the same for independent management parties in *MP2*. Ideally, this would mean that wherever in the above formulas M_1 or M_2 are written, 1 could be written instead. However, the reality is more complex, because some management parties might need finalization information from other management parties.

I verified these and other formulas on a number of example scenarios by comparing numbers generated by formulas and numbers that I counted on UML sequence diagrams I draw for these scenarios. For example, when all monitoring is performed by one inde-

pendent accounting party, then A_1=1, M_1=0, C_1=0, A_2=1, M_2=0, C_2=0, $D_{1:2}$=0, $E_{1:2}$=1, $S_{1:2}$=0, so consumer-initiated switching without any optimization requires:

$2*(2+3*A_1+M_1+C_1+A_2+M_2+D_{1:2}) = 12$ SOAP messages.

However, when protocol is not modularized, initialization and finalization are combined, and the provider is not explicitly informed about the results of finalization, then the savings are:

$2*(1+1*A_1)+2*(S_{1:2}+E_{1:2})+2*(A_1) = 8$ SOAP messages.

The difference is: 12-8 = 4 SOAP messages.

The example in Figure 4.1 used the above mentioned conditions and optimizations. The number of exchanged SOAP messages is indeed 4.