Passive Observation-Based Architectures for Management of Web Services

Abdelghani Benharref

A thesis
In
the Department
of
Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy at
Concordia University
Montréal, Québec, Canada

November 2007

© Abdelghani Benharref, 2007

Library and Archives Canada

Published Heritage Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et Archives Canada

Direction du Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 978-0-494-37750-5
Our file Notre référence
ISBN: 978-0-494-37750-5

NOTICE:
The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:
L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Abstract

Passive Observation-Based Architectures for Management of Web Services

Abdelghani Benharref, Ph.D.
Concordia University, 2007

Web Services technologies are emerging as the standard paradigm for program-to-program interactions over the Internet. A Web Service is any application that offers its functionalities through the Internet by publishing a description of its interfaces. Web Services are gaining more and more momentum and their utilization is being spread and even standardized in many areas including e-Government, e-Telecomm, e-Health, and digital imaging.

The management of Web Services will play an important role for the success of this emerging technology and its adoption by both providers and consumers. As the technology matures and spreads, consumers are likely to be very picky and restrictive with regards to the quality of the offered Web Services. Another challenging factor for the management of Web Services is related to the diversity of platforms on which Web Services are developed and deployed.

In this thesis, the focus is on the management of Web Services using passive observation with the intent to have open and platform-independent management architectures capable of assessing both functional and non-functional aspects of Web Services. The bulk of the observation process is carried out by model-based entities known as observers. These observers make use of formal model such as Finite State Machines, Communicating Finite State Machines, and Extended Finite State Machines.

The proposed architectures include observers developed and deployed as Web Services: mono-observer architecture and multi-observer architectures. A single observer is enough for observation of a non-composite Web Service while a network of observers is preferred when observing a composite Web Service. Passive observation requires traces' collection mechanisms which are thoroughly studied and their performance compared for all architectures.

A new approach for online observation based on Extended Finite State Machine is proposed to accelerate misbehaviors' detection. This approach proposes backward and forward walks in the model to reduce possible sets of states and values of variables.

I adopted a pragmatic evaluation approach to assess each of my contributions: analytical analysis and proof, implementation, and real case studies. All components of management architectures have been studied, their complexities determined, developed, and deployed. The use cases used for the evaluation of the effectiveness of the architecture, including simple and composite Web Services, are also fully implemented and deployed.

Acknowledgments

I no doubt deserved my enemies, but I don't believe I deserved my friends - Walt Whitman

This thesis, as simple as it is, could not be accomplished without contributions of a nice cohort of lovely partners. I will not be able to name all of them since this would require dozens of pages; I start by thanking all those that I will not explicitly name hereafter.

Since the early beginning, Rachida[1], my supervisor, provided an invaluable and unforgettable support: scientific, moral, and financial. Her encouragements and motivations during those (multiple) toughest moments were crucial and of prime importance. I hereby thank her for all what she did for me, even though I will never be able to express enough thanks and gratefulness. I will also take this opportunity to think my co-supervisor, Dr. Roch Glitho.

I thank my colleagues at Concordia for their help and long-running discussions. I have been very fortunate to be surrounded by a group of nice and careful ladies: Rabeb, May, Fatna, Syrine, and Yosr and their families. Dish-parties we organized at some occasions presented a nice shift from boring research to funny life. I thank as well Mohamed Adel Serhani, Abdeslam En-Nouaary, Mohamed Vall Ould Zein, Jamal Bentahar, and Mohamed El Hachimi for the priceless support, discussions, and experiences we have shared.

[1] Since the first time we met for my master, I used to call her "Rachida" rather than the official name "Dr. Dssouli". I prefer to stick to that even in formal thesis acknowledgments. That looks friendlier and reflects the kind of relationship we had, have, and will have in the future.

Thanks are extended to my food-oriented friends: Sanaa and Omar Daoudi, Hicham Achir, Myriam and Chakib Azizi, Kirsten and Mustapha Labiz, Saaida and Hamza, and to their respective families.

These acknowledgments will neither be complete, fair, nor honest without expressing my deepest thanks and gratitude to Dalia Radwan. She has been very attentive, supportive, and exceptionally helpful during the long-lasting years of this Ph.D.

Abdelghani Benharref
November, 2007

إهداء

قبل كتابة هذه الكلمات، كنت قد وضعت حدا لمسيرة دامت زهاء ربع قرن من طلب العلم و التحصيل بمؤسسات مختلفة. طول المدة لا يتوافق بالضرورة و كمية أو جودة المعلومات المحصل عليها؛ بيد أنه يخفي وراء ظلاله صبرا و مثابرة ما كان لي أن انجبهما من تلقاء نفسي. هذا الصبر و هذه المثابرة، دون أن يساورني ادنى شك، هو نتاج جهود متظافرة لمجموعة لا بأس بها من أفراد عائلة بنحرف المتشعبة وثلة من أصدقائها. أتوجه هنا بالشكرالى كل هؤلاء.

بادئ ذي بدء، أخي سي محمد. هذا الأخ العزيز الذي لم يبخل عليّ يوما بماله و لا بوقته و لا بنصائحه، وذلك مند زمن بعيد. وكم كان مشرفا لي أن يعبر المحيط الأطلسي ذهابا و إيابا في غضون أسبوعين لحضور مناقشة هذه الأطروحة. أرجو أن يجد بين طيات هذه الكلمات بعضا من معاني الشكر العميق و الإعتراف بالجميل.

أتوجه بالشكرالى أخي سي مصطفى. أخ دأب جاهدا على مساعدتي كلما دعت الضرورة لذلك، ماديا ومعنويا.

كذلك حال أخي بالمصاهرة، عبد الرحيم قيبوس وباقي الاخوة و الأخوات: للا فاطنة، للا أمينة، للا مباركة، عبد الهادي، فتيحة، خديجة، ربيعة، عزيز، لبنى، سهام، أحلام، سي محمد، ابتسام، و سي محمد أمين.

أبعث بالشّكر إلى كل أفراد العائلة و الأصدقاء بكل من الرباط، تمارة، الصخيرات، الدر البيضاء، الجديدة، سيدي بنور، و الصويرة.

و أخيرا، شكر خالص للأبوين، سبب وجودي بهذه الحياة، مع كلّ ما لذلك من سلبيات وايجابيات.

ويستمر طلب العلم و التحصيل بمدرسة الحياة...

عبد الغني بن علي بن علي

تشرين الثاني ٢٠٠٧،

يا أيها الانسان ما غرّك بربك الكريم، الذي خلقك فسوّاك فعدّلك، في أيّ صورة ما شاء ركّبك.

سورة الانفطار، آيات 6 الى 8.

Ô homme ! Qu'est-ce qui t'a trompé au sujet de ton Seigneur, le Noble, qui t'a créé puis modelé et constitué harmonieusement? Il t'a façonné dans la forme qu'Il a voulue.

Sourate Al-Infitar (La rupture), versets 6 à 8.

O you human being! What diverted you from your Lord Most Honourable? The One who created you, designed you, and perfected you. In whatever design He chose, He constructed it.

Sura Al-Infitar (The Shattering), verses 6 to 8.

Table of Contents

TABLE OF CONTENTS X

LIST OF TABLES XIV

LIST OF FIGURES XV

LIST OF ACRONYMS XVII

LIST OF ALGORITHMS XX

CHAPTER 1 INTRODUCTION 1

1.1 MOTIVATIONS AND OBJECTIVES OF THE THESIS 3

1.2 THESIS CONTRIBUTIONS 5

1.3 THESIS ORGANIZATION 6

CHAPTER 2 BACKGROUND INFORMATION 8

2.1 SERVICE ORIENTED ARCHITECTURE 9

2.1.1 Roles *9*

2.1.2 Operations *10*

2.2 PROGRAMMING STACK 11

2.3 EXAMPLE OF WEB SERVICES 14

2.4 MANAGEMENT AREAS 17

2.4.1 Fault management *17*

2.4.2 Configuration management *18*

2.4.3 Accounting management *18*

2.4.4 Performance management *18*

2.4.5 Security management *18*

2.5 TESTING FOR FAULT DETECTION 19

2.5.1 Active testing *20*

2.5.1.1 Test cases extraction 20

2.5.1.2 Test cases execution 20

2.5.1.1 Results and analysis 21

2.5.2 Passive testing *22*

2.5.2.1 FSM 24

2.5.2.2 CFSM 25

2.5.2.3 EFSM 26

2.6 EXAMPLE OF FAULT DETECTION BY PASSIVE OBSERVATION 27

2.7 SUMMARY 29

CHAPTER 3 RELATED WORK 31

3.1 WEB SERVICES MANAGEMENT 31

3.1.1 Functional management *34*

3.1.2 QoWS management *35*

3.1.3 Discussion *37*

3.2 PASSIVE TESTING 38

3.3 SUMMARY 41

CHAPTER 4 EXTENDING SOA WITH OBSERVATION CAPABILITIES 43

4.1 EXTENDED SOA 44

4.1.1 ESOA Features *44*

4.1.2 Procedure and Components *46*

4.1.3 Number of observers and location of points of observation *47*

4.1.4 Requirements *50*

4.2 SUMMARY 51

CHAPTER 5 WEB SERVICES' BEHAVIOR DESCRIPTION 52

5.1 WSDL 54

5.2 DESCRIPTION OF THE FUNCTIONAL ASPECTS 57

5.2.1 FSM *58*

5.2.2 EFSM 58
5.2.3 BPEL 60
5.3 QoWS ASPECTS 64
5.3.1 QoWS attributes 64
5.3.2 QoWS attributes in a dedicated document 67
5.3.3 QoWS specification embedded with functional behavior 69
5.3.4 QoWS attributes in WSDL 69
5.4 SUMMARY 71

CHAPTER 6 MANAGEMENT ARCHITECTURE FOR SIMPLE WEB SERVICES 73

6.1 COMMUNICATION BETWEEN COMPONENTS 73
6.1.1 Client/Web Service instrumentation 74
6.1.2 Multicast 75
6.1.3 Dispatcher 76
6.1.4 SNMP 77
6.1.5 Mobile code 79
6.1.6 Discussion 80
6.2 CASE STUDY 81
6.2.1 Web Service Observer 84
6.2.2 Client 85
6.2.3 Detection capabilities 86
6.2.4 Quantitative evaluation of the architecture 88
6.2.4.1 Pre-observation configurations 88
6.2.4.2 Processing CPU and memory utilization 93
6.2.4.3 Network overhead 94
6.2.5 Discussion 96
6.3 SUMMARY 97

CHAPTER 7 OBSERVATION ARCHITECTURES FOR COMPOSITE WEB SERVICES 99

7.1 PROCEDURE 100
7.2 NUMBER AND LOCATIONS OF MOBILE OBSERVERS 101
7.2.1 Composite Web Service provider's participation 103
7.2.2 Basic Web Services providers' participation 105
7.2.3 Hybrid participation 106

7.2.4 Discussion *108*

7.3 EXAMPLES OF OBSERVERS' ALGORITHMS 109

7.4 CASE STUDY 114

7.4.1 Case study description *115*

7.4.2 Web Services *116*

7.4.3 Implementation issues *117*

7.4.4 Single observation limitations *119*

7.4.5 Example of multi-observer observation procedure *121*

7.4.6 Network load *125*

7.4.6.1 Deployment load 125

7.4.6.2 Traces' collection load 126

7.4.7 Results and analysis *127*

7.5 SUMMARY 130

CHAPTER 8 EFSM-BASED OBSERVATION 132

8.1 EFSM-BASED OBSERVERS: FORWARD AND BACKWARD WALKS 133

8.1.1 Homing controller procedure *135*

8.1.2 Processing requests *137*

8.1.3 Processing responses *139*

8.1.1 Performing backward walks *141*

8.2 DISCUSSION 144

8.3 EXAMPLE 147

8.3.1 Observation without backward walks *148*

8.3.2 Observation with backward walks *148*

8.4 SUMMARY 149

CHAPTER 9 CONCLUSION AND FUTURE WORK 151

9.1 SUMMARY OF THESIS 151

9.2 THESIS CONTRIBUTIONS 152

9.3 FUTURE WORK 155

BIBLIOGRAPHY 157

List of Tables

TABLE 6.1. EXPOSED FUNCTIONALITIES AND THEIR CORRESPONDING PUBLISHED INTERFACES 82

TABLE 6.2 EXECUTED SCENARIOS AND THEIR CORRESPONDING VERDICTS 87

TABLE 6.3. CHARACTERISTICS OF TRACES' COLLECTION MECHANISMS 97

TABLE 7.1 SOME OF THE EXECUTED SCENARIOS 128

TABLE 8.1 CONTENT OF SPS, SPVV, SKV, SUV, AND TPPS 149

List of Figures

FIGURE 2.1. SERVICE ORIENTED ARCHITECTURE 10
FIGURE 2.2 SOA PROGRAMMING STACK 12
FIGURE 2.3 INVOKING A WEB SERVICE TO COMPUTE N! 14
FIGURE 2.4 XML-BASED MESSAGING IN SOA 16
FIGURE 2.5 STANDARD ACTIVE TESTING ARCHITECTURES 21
FIGURE 2.6 PASSIVE TESTING ARCHITECTURE 22
FIGURE 2.7 SPECIFICATION AND IMPLEMENTATION FSM MACHINES 28
FIGURE 4.1 ESOA WITH PASSIVE OBSERVATION 46
FIGURE 4.2. MONO-OBSERVER ARCHITECTURE 48
FIGURE 4.3. MULTI-OBSERVER ARCHITECTURE 49
FIGURE 5.1 EXTENDING THE USE OF FORMAL MODELS BY/FOR OBSERVATION 53
FIGURE 5.2 PARTIAL WSDL OF THE CALL CONTROL WEB SERVICE 56
FIGURE 5.3 XML REPRESENTATION OF AN FSM MACHINE 59
FIGURE 5.4 XML REPRESENTATION OF AN EFSM MACHINE 60
FIGURE 5.5 STRUCTURE OF BPEL PROCESS 61
FIGURE 5.6 BPEL PROCESS WITH PARTNERLINKS 62
FIGURE 5.7 EXAMPLE OF BPEL ACTIVITIES TAG 63
FIGURE 5.8 QOWS SPECIFICATION IN DEDICATED DOCUMENT 68
FIGURE 5.9 FSM+ SPECIFICATION 70
FIGURE 5.10 QOWS IN WSDL DOCUMENT 71
FIGURE 6.1 CLIENT'S INSTRUMENTATION FOR TRACES' COLLECTION 75
FIGURE 6.2 MULTICAST FOR TRACES' COLLECTION 76
FIGURE 6.3 SNIFFERS-DISPATCHERS FOR TRACES' COLLECTION 77
FIGURE 6.4 SNMP AGENTS FOR TRACES' COLLECTION 78
FIGURE 6.5 XML DESCRIPTION OF THE CALL CONTROL FSM MACHINE 83

FIGURE 6.6 GRAPHICAL REPRESENTATION OF THE CALL CONTROL FSM MACHINE 84
FIGURE 6.7 CLIENT APPLICATION GUI 86
FIGURE 6.8 TCP-DISPATCHER CONFIGURATION 90
FIGURE 6.9 TCP-BASED DISPATCHING 91
FIGURE 6.10 SNMP AGENT TRACES' COLLECTOR 92
FIGURE 6.11 MOBILE OBSERVER GUI 93
FIGURE 6.12 NETWORK LOAD MEASURES FOR TRACES' COLLECTION MECHANISMS 96
FIGURE 7.1 COMPOSITE WEB SERVICE'S PROVIDER PARTICIPATION 104
FIGURE 7.2 ALL PROVIDERS' PARTICIPATION 106
FIGURE 7.3 HYBRID PARTICIPATION 107
FIGURE 7.4 COMPOSITE/BASIC WEB SERVICES 117
FIGURE 7.5 SINGLE-OBSERVER CONFIGURATION 120
FIGURE 7.6 SINGLE-OBSERVER DEPLOYMENT 121
FIGURE 7.7 MULTI-OBSERVER CONFIGURATION 123
FIGURE 7.8 MULTI-OBSERVER DEPLOYMENT 124
FIGURE 7.9 MULTI-OBSERVER ARCHITECTURES ASSOCIATED NETWORK OVERHEAD 127
FIGURE 7.10 TRACE LOST BEFORE GETTING TO LOCAL OBSERVER 129
FIGURE 7.11 TRACE OR FAULT NOTIFICATION LOST BEFORE GETTING TO GLOBAL OBSERVER 130
FIGURE 8.1 CONSTRAINTS PROPAGATION 143
FIGURE 8.2 SPVV AND CONSTRAINTS PROPAGATION 146
FIGURE 8.3 EFSM EXAMPLE 147
FIGURE 8.4 TPPS 150
FIGURE 9.1 CHRONOLOGY OF THESIS CONTRIBUTIONS 153

List of Acronyms

AMD	Advanced Micro Device
API	Application Programming Interfaces
ASP	Active Server Pages
BPEL	Business Process Execution Language
CC	Call Control
CFSM	Communicating Finite State Machines
CGI	Common Gateway Information
CMIP	Common Management Information Protocol
CMOT	CMIP Over TCP/IP
CPU	Central Processing Unit
CPXe	Common Picture eXchange environment
CWS	Conferencing Web Service
EFSM	Extended Finite State Machine
ESOA	Extended Service Oriented Architecture
FCAPS	Fault, Configuration, Accounting, Performance, and Security
FIFO	First In First Out
FSM	Finite State Machine
FSM+	QoS-annotated FSM
FTP	File Transfer Protocol
GUI	Graphical User Interface
HTTP	HyperText Transfer Protocol
ID	Identifier
IF	Input Faults
IP	Internet protocol
ISO	International Organization for Standardization

ITF	INPUT TYPE FAULT
IUT	IMPLEMENTATION UNDER TEST
JADE	JAVA AGENT DEVELOPMENT FRAMEWORK
JSP	JAVA SERVER PAGES
LT	LOWER TESTER
MIB	MANAGEMENT INFORMATION BASE
MNPT	MINIMUM PROCESSING TIME
MNRT	MINIMUM RESPONSE TIME
MXPT	MAXIMUM PROCESSING TIME
MXRT	MAXIMUM RESPONSE TIME
OF	OUTPUT FAULTS
OSI	OPEN SYSTEM INTERCONNECTION
OTF	OUTPUT TYPE FAULT
PCO	POINT OF CONTROL AND OBSERVATION
PO	POINT OF OBSERVATION
PT	PROCESSING TIME
QOS	QUALITY OF SERVICE
QOWS	QUALITY OF WEB SERVICE
RAM	RANDOM ACCESS MEMORY
RT	RESPONSE TIME
SC	SERVICE CHARGE
SKV	SET OF KNOWN VARIABLES
SLA	SERVICE LEVEL AGREEMENT
SMTP	SIMPLE MAIL TRANSFER PROTOCOL
SNMP	SIMPLE NETWORK MANAGEMENT PROTOCOL
SOA	SERVICE ORIENTED ARCHITECTURE
SOAP	SIMPLE OBJECT ACCESS PROTOCOL
SPS	SET OF POSSIBLE STATES
SPVV	SET OF POSSIBLE VARIABLE VALUES
SUO	SYSTEM UNDER OBSERVATION
SUV	SET OF UNKNOWN VARIABLES
TCP	TRANSFER CONTROL PROTOCOL
TLTS	TIMED LABELED TRANSITION SYSTEMS

TPG	TASK PRECEDENCE GRAPH
TPPS	TREE OF POSSIBLE PREVIOUS STATES
UDDI	UNIVERSAL DESCRIPTION, DISCOVERY, AND INTEGRATION
UDP	SER DATAGRAM PROTOCOL
UT	UPPER TESTER
VPN	VIRTUAL PRIVATE NETWORK
WS	WEB SERVICE(S)
WSCI	WEB SERVICES CHOREOGRAPHY INTERFACE
WSCL	WEB SERVICES CONVERSATION LANGUAGE
WSDL	WEB SERVICE DESCRIPTION LANGUAGE DOCUMENT
WSFL	WEB SERVICES FLOW LANGUAGE
WSO	WEB SERVICE OBSERVER
WSUO	WEB SERVICE UNDER OBSERVATION
WSUT	WEB SERVICE UNDER TEST
WSUT	WEB SERVICE UNDER TEST
XML	EXTENSIBLE MARKUP LANGUAGE
XSD	XML SCHEMA DATATYPES

List of Algorithms

Algorithm 7.1 Misbehavior detection 110

Algorithm 7.2 Global purge 111

Algorithm 7.3 Local notification and purge 112

Algorithm 7.4 Correlation 113

Algorithm 8.1 Homing controller 136

Algorithm 8.2 Processing observed requests 138

Algorithm 8.3 Processing observed responses 140

Algorithm 8.4 Performing backward walks 142

Chapter 1
Introduction

Perfection is achieved, not when there is nothing more to add, but when there is nothing left to take away - Antoine de Saint-Exupery.

The past decade saw a dramatic change in the nature of communication and interactions over the Internet. In the early days of Internet, interactions were simply based on basic HyperText Transfer Protocol (HTTP) functionalities and were limited to the exchange of static documents between clients and servers. Later on, dynamic content became available through the introduction of many scripting languages which enhanced the capabilities of web servers. Plug-ins such as Common Gateway Information (CGI), Java Server Pages (JSP), and Active Server Pages (ASP) enabled dynamic contents provisioning through gateways between clients, web servers, and backend database servers. While this kind of interactions is still widely used by industry, a new approach for communication and access to content over the Internet, known as *Web Services*, is nowadays emerging.

The concept of Web Services stands for a new generation of Web applications. It is a collection of mechanisms that allows automatic communication between applications through the Internet. Web Services operations are structurally described and published for eventual use by interested applications. This new paradigm of communication puts more emphasize on business-to-business interactions while still supporting the business-to-consumer transactions model that the Internet is largely providing. Another attracting fea-

ture in the new paradigm is the composition of Web Services as opposed to the build-from-scratch approach for developing Web Services; existing Web Services are used by aggregating their functionalities to provide a richer and complete composite Web Services with wider functionalities.

A considerable effort is undergoing to promote and spread the use of Web Services. Indeed, both the industry and academia were, and still, are working extensively to provide automated processes for the development, use, and management of Web Services. The major concerns are the expressiveness of Web Services descriptions, Web Services' languages, protocols and architectures.

The management of Web Services is of prime importance for all entities involved in Web Services industry. The management is quite different from the traditional management of distributed systems in the new environments where interacting components may not be known a priori, may be on different operating systems and platforms and often implemented using different programming languages. The management functions handle issues such as fault, configuration, accounting, performance, and security. The fault management function in particular includes sub-functionalities for the detection, localization, isolation and repair of faults (faulty behaviors).

When I started the research for my thesis, the emphasis in Web Services technology was on their development rather than on management, except for basic key features (deploy, undeploy, etc.), which were naturally included within hosting Web Services platforms. Advanced fault and performance management operations such as monitoring and third party testing/certification for instance, were not considered. Although, it was only a matter of time

as it is evident that clients/users will favor only the use of Web Services known to behave correctly and which provide good performance. Consequently, fault and performance management will play a key role in selecting and using Web Services.

This thesis presents a series of contributions on management of synchronous Web Services. It focuses more on fault and performance management but provides also hints for accounting. Accounting management is supported by defining Service Charge (SC) for each invoked operation and selected profiles. For fault and performance management, functional and non-functional violations are detected, failed Web Services are identified and located. Isolation and repair are left for future work.

The main objective of these contributions is to move management operations away from the platforms on which Web Services are deployed. This goal is achieved through the design of platform-independent and open architectures suitable to tackle the different management issues. The contributions of this thesis are discussed in more details further in section 1.2.

1.1 Motivations and objectives of the thesis

Nowadays, management of Web Services is highly platform-dependent. Most Web Services hosting platforms, in addition to their server capabilities, offer features to manage Web Services. For example, WebSphere [1], BEA WebLogic [2] and Microsoft .Net [3] all come with a set of management utilities that must be installed in addition to the core Web Services' hosting platform.

Incorporating management tools into the hosting platforms restricts their use to providers of Web Services. Clients (i.e. users) as well may need access to these management fea-

tures for various reasons. A client for example might need to check the correctness of a Web Service she/he is actually using or planning to use in the future.

There are actually very few management tools that operate outside hosting platforms. Examples of such tools include SOATest [4], .Test [5], and Webking [6]. However, these tools are based on active testing which include the generation of test cases, their execution, and the analysis of responses received from a Web Service. Such an approach does not allow online verification of the interactions between a Web Service and its client since it requires the generation of a set of appropriate test cases and their application.

The review and analysis of the existing solutions for management of Web Services led to the belief that one needs first to consider the following main questions before defining a suitable Web Services management approach:

- How to manage a Web Service in its final environment without disturbing its normal operation?
- How to conduct this "non-disturbing" management online and in an automatic manner?
- How to make the management system available to all involved entities (client, provider, and any potential third party)?
- How to minimize required resources to use the management system?

The detection of misbehaviors without disturbing an already deployed Web Service requires the so-called *passive testing*. In this kind of testing, an *observer* checks passively exchanged messages (traces) between a Web Service and its client and assesses their compliance with an expected behavior. A key concern in passive testing is the collection of traces

and how to convey passively traces to an observer. The number and location of observers are also important issues to be considered in passive testing especially in the case of composite Web Service.

The next section presents the contributions of the thesis, which address the above raised issues.

1.2 Thesis contributions

The central idea behind my research is the conception, design, and development of new (passive) observation architectures dedicated to management of synchronous Web Services. A wider range of other research issues related to these architectures and testing in general were identified, investigated, and solutions are proposed and evaluated. The need for new architectures for management is motivated by the emergence and the maturity of Web Services technologies and their appealing benefits. The following sections present a high level overview of my contributions; the details of each contribution will be presented in the subsequent chapters of the thesis.

The first contribution consists of a proposal for the extension of the Service Oriented Architecture (SOA) with observation capabilities by designing observers in the form of Web Services achieving thus, an open, platform-independent and loosely coupled management approach.

The second contribution consists of an observation architecture for functional management of a non-composite Web Service. The effectiveness of this architecture is evaluated as well as the performance of a set of mechanisms for traces' collection.

The third contribution consists of an extension and enhancement of the previous architecture for the observation of composite Web Services where networks of observers are used to improve misbehaviors detection capabilities. Quality of Service's aspects are also considered as well as functional aspects. The number of observers, their locations, and location of points of observation affects the effectiveness and the overhead of the architecture. Based on these factors, three multi-observer architectures are proposed and studied.

Finite State Machine (FSM) models do not support data flow, which is considered in the EFSM models. The last contribution of this thesis proposes new algorithms to enhance EFSM-based observers. This enhancement is made possible by performing backward walks whenever the observer is waiting for exchanged messages.

The new EFSM observer-based approach can be used for the observation of any distributed system. Other non EFSM-based approaches described in the thesis are however, limited to the observation of Web Services as they are based on Web Services-specific protocols and technologies not necessarily available outside SOA.

1.3 Thesis organization

This thesis is organized as follows:

Chapter 2 gives the reader background information to follow and understand the flow of ideas in the upcoming chapters. The background is mainly focused on the two main concepts of this thesis namely *Web Services' management* and *passive testing*.

Related work on Web Services' management is described in Chapter 3. This chapter presents and discusses different existing approaches for management of Web Services. The chapter also includes a summary of related work on passive testing.

The rest of the thesis (from Chapter 4 to Chapter 8), describes my contributions. Extension of the SOA with passive observation capabilities is described in Chapter 4. This extension proposes two novel architectures for management of Web Services: mono-observer architecture for basic Web Services and a multi-observer architecture for composite Web Services.

As a requirement of these architectures, the expected behavior of the Web Service Under Observation (WSUO) must be handed to the observer before any observation activities can take place. In Chapter 5, I discuss how the expected behaviors could be specified including both functional and non-functional aspects.

Chapter 6 outlines our first experimentation with the mono-observer architecture. First, a set of mechanisms for traces' collection and communication between entities of the architecture is discussed. The effectiveness and the overhead of the architecture are then evaluated through a case study.

Chapter 7 illustrates the use of multi-observer architectures to observe a composite Web Service.

Chapter 8 presents new efficient algorithms for EFSM-based observers by combining both observed traces and backward walks of the Web Service's EFSM.

Chapter 9 concludes this dissertation and recall briefly its contributions. It elaborates also on guidelines for future work.

Chapter 2
Background Information

The ability to simplify means to eliminate the unnecessary so that the necessary may speak -
Hans Hofmann

This chapter provides essential background information on Web Services and passive testing required for a complete understanding of upcoming chapters.

Web Services are a new variant of web applications. It is a new paradigm that allows making various applications communicate with each other automatically over the Internet. They are self-contained, self-describing, modular applications that can be published, located, and invoked across the Internet[7]. The goal is to allow applications to be delivered over the Internet and run across all kinds of computers and platforms.

A Web Service is any application that can be published, located, and invoked through the Internet. Each Web Service has a Web Service Description Language document (WSDL)[8], which is an XML [9] document providing all the required knowledge to communicate with the Web Service including its location, supported transport protocols, messages formats, and list and signatures of published operations.

A Web Service can perform any kind of transactions that may range from getting a city's temperature to a more complicated transaction like for instance searching and/or building the best travel packages from specific travel agencies. The main objective of Web Services is to allow, at a high level of abstraction, applications to be accessible over the Internet. They

can be of great use, for example, for 3G networks operators to expose their core network functionalities to third parties [10]. Digital imaging is another field where Web Services can make an important benefit to the digital photography industry. The Common Picture eXchange environment (CPXe) [11], a Web Service business framework, will allow transfer and printing of digital images as suitable as using films.

2.1 Service Oriented Architecture

The Service Oriented Architecture [12] defines three roles (service provider, service requester, and service registry) and three operations (publish, find, and bind). The relationship between the roles and the operations are illustrated in Figure 2.1. Additional information on the Web Services architecture can be found in [13].

2.1.1 Roles

- **Service Provider:** is the owner of the Web Service and its hosting platform
- **Service Requestor:** represents the client, i.e., the entity which makes use of the Web Service provided by the Web Service's provider
- **Service Registry:** this is a yellow pages-like database of Web Services' descriptions. A client interested in some kind of Web Services can query this database to get a list of potential Web Services satisfying its request. Each record in this database contains all the required information to use the target Web Service.

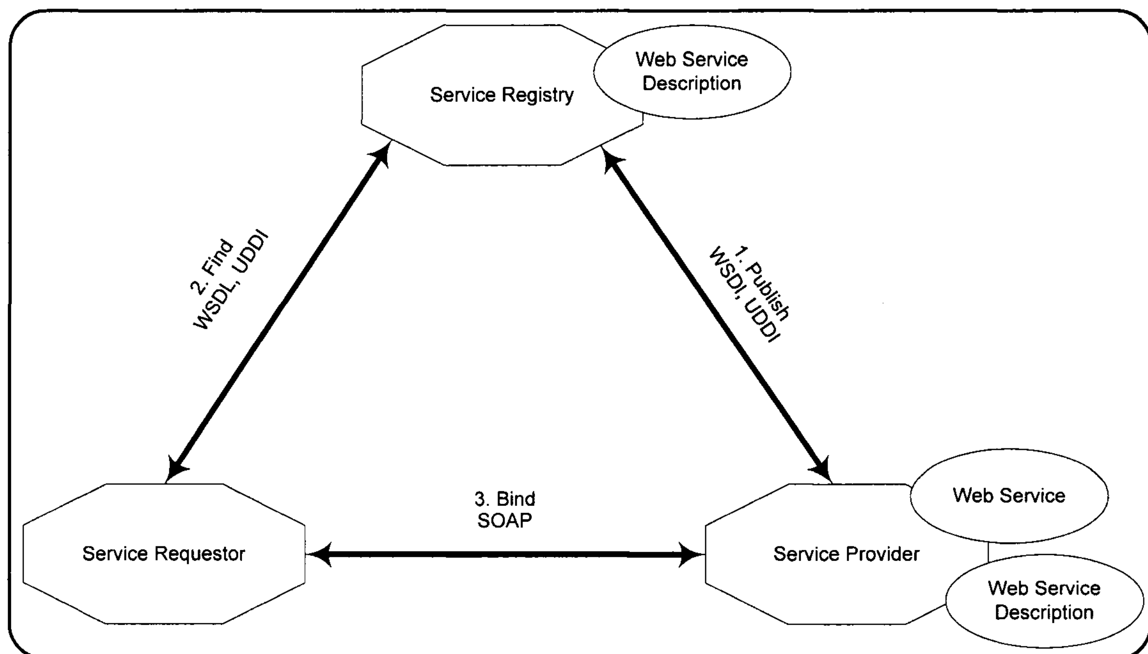

Figure 2.1. Service Oriented Architecture

2.1.2 Operations

- **Publish**: in order to be known by Web Services' requestors, the WSDL document of a Web Service must be published. Publication can be any action that makes the Web Service description document available for requestors. It ranges from static publication using email or File Transfer Protocol (FTP) to dynamic using Universal Description, Discovery, and Integration (UDDI) registry[14].
- **Find**: the nature of this operation depends on the publication method used to publish the desired Web Service's description. The find operation may be as simple as a quick search in an email inbox if the Web Service is published using emails. It may however be more complex and necessitates the use of a UDDI reg-

istry to deal with dynamic publication. The find operation is used to locate and return the Web Service's description to a requestor. It is executed in two steps in the Web Service life cycle: during design of the Web Service's client to retrieve the Web Service interfaces' description and during runtime to retrieve the Web Service's binding and location information.

- **Bind:** this operation is used to invoke operations from the desired Web Service.

2.2 Programming stack

The Web Services programming stack (Figure 2.2) is a collection of standardized protocols and Application Programming Interfaces (API) allowing Web Services' location and utilization. The stack consists of 6 layers and 3 towers. At each layer, open protocols are already standardized or work is in progress towards their standardization. The towers define features that should be present in all layers: *Security*, *management*, and *Quality of Service*. The three lower layers are required for Web Services interoperability while the 3 higher layers are optional and used if needed.

- **Network:** The Network Layer is the basis of the Web Services programming stack. A Web Service must be available over a network. This layer is often based on HTTP due to its widespread utilization although other protocols might be used.
- **XML-Based Messaging:** On the top of the network layer, this layer uses Simple Object Access Protocol (SOAP) to permit communications between Web Services and their clients.

- **Service Description**: The description of the interfaces and supported protocols for Web Services' interactions are the minimal requirements for Web Services communication. Other requirements can also be specified including Quality of Service or security constraints. This description consists of WSDL documents.

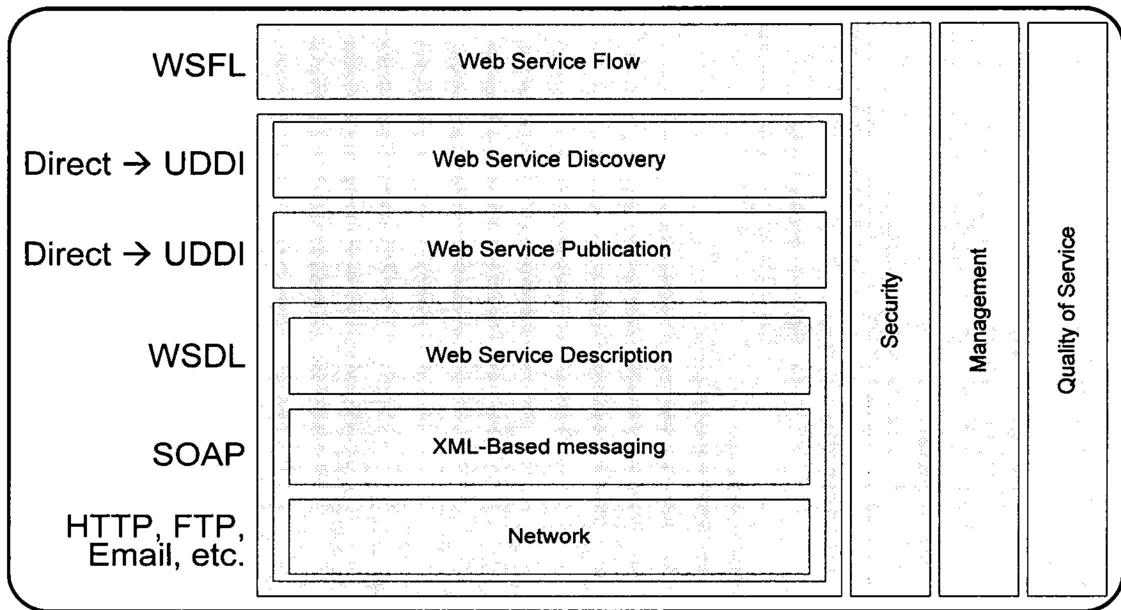

Figure 2.2 SOA programming stack

- **Service Publication**: For a Web Service to be known by clients, its description should be published. Service publication can be any action that lets clients know about the Web Service. E-mail and UDDI registries are examples of possible mechanism for Web Services publication.
- **Service Discovery**: Depending on how Web Services publication has been performed, Web Services discovery is any action that returns the WSDL service description to a Web Service requestor.

- **Service Flow**: This layer facilitates the composition of Web Services into workflows and the representation of this aggregation to a higher-level Web Service.

The starting point in Web Services interactions is the development, deployment, and publication of Web Services. When a client needs a specific Web Service, she/he probes the UDDI registry for a list of potential providers. The returned list contains matching records; each record contains required information to connect to the corresponding Web Service. Based on some criteria (location, availability, etc), the client selects the suitable Web Service and invokes it. Section 2.3 presents an example of invocation of a Web Service to compute the factorial of an integer.

Web Services can be developed either from scratch or by composition. Composition of Web Services is the process of aggregating a set of Web Services to create a more complete Web Service with a wider range of functionalities. This composition has a considerable potential of reducing development time and effort for new applications by reusing already available Web Services.

Currently, there are standards or languages that help building composite Web Services such as: Web Services Flow Language (WSFL)[15], DAML-S [16], Web Services Conversation Language (WSCL) [17], Web Services Choreography Interface (WSCI) [18], and Business Process Execution Language (BPEL) [19]. These languages make easier Web Services composition process by providing concepts to represent partners and orchestrate their interactions. BPEL, which represents the merging of IBM's WSFL and Microsoft's XLANG, is gaining a lot of interest and is positioned to become the primer standard for Web Service composition.

2.3 Example of Web Services

A client has to compute the n! (n factorial), but does not have the required processing capabilities. Fortunately, she/he has been told that some Web Services providers are offering this Web Service through the Internet. In Figure 2.3.a, three providers are publishing their Web Services that calculate the factorial of an integer to a UDDI registry. The client looks in the UDDI registry for potential Web Services, and gets a list containing the descriptions of the 3 Web Services published earlier (Figure 2.3.b). For the client, Web Service's provider 2 seems to satisfy her/his needs so she/he decides to use it (Figure 2.3.c).

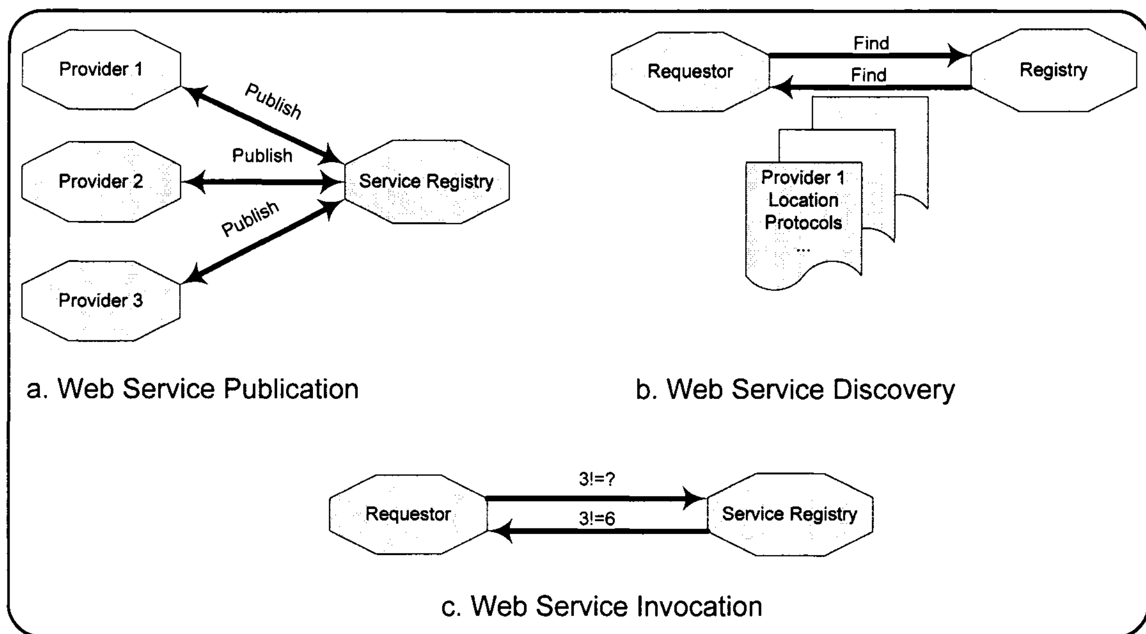

Figure 2.3 Invoking a Web Service to compute n!

All communications between Web Services entities are based on XML and use SOAP. SOAP is basically an HTTP POST with an XML envelope as a payload. It defines a simple mechanism for expressing application semantics by providing a modular packaging model

and mechanisms for encoding data within modules [20]. It can be used with a variety of protocols such as HTTP, Simple Mail Transfer Protocol (SMTP), and FTP. Figure 2.4 shows how XML messaging based on SOAP and network protocols form the basis of the SOA. It is to be noted that the statement "network protocols" in Web Services context is different from the one used within the Open System Interconnection (OSI) ([21]) and the Internet Protocol (IP) ([22]) architectures. Network protocols in Web Services architectures stand for application protocols capable of enveloping and transporting SOAP messages exchanged by Web Services. Protocols such as SMTP and FTP may be used but HTTP is the most considered because it is now the only protocol for which a standard binding for SOAP exists.

The ability to build and/or parse SOAP messages and to communicate over a network is the basic requirements for a network node to play the role of Web Service requester or provider in XML messaging-based distributed computing [13]. The chronology of these interactions, numbered 1, 2, 3 and 4 in Figure 2.4, between a Web Service and a requester, is as follows:

1. The first step in the interaction between a requestor and a Web Service is the formulation of the request. The requestor must know the message format to use to correctly communicate with the Web Service. All information required for communication between the requestor and the Web Service is contained within the WSDL document. The requestor builds a SOAP message containing the request. The SOAP message is presented, with the network address of the Web Service provider, to the SOAP infrastructure (for example, a SOAP client run-

time). The SOAP infrastructure interacts with the underlying network protocol to send the request out over the network.

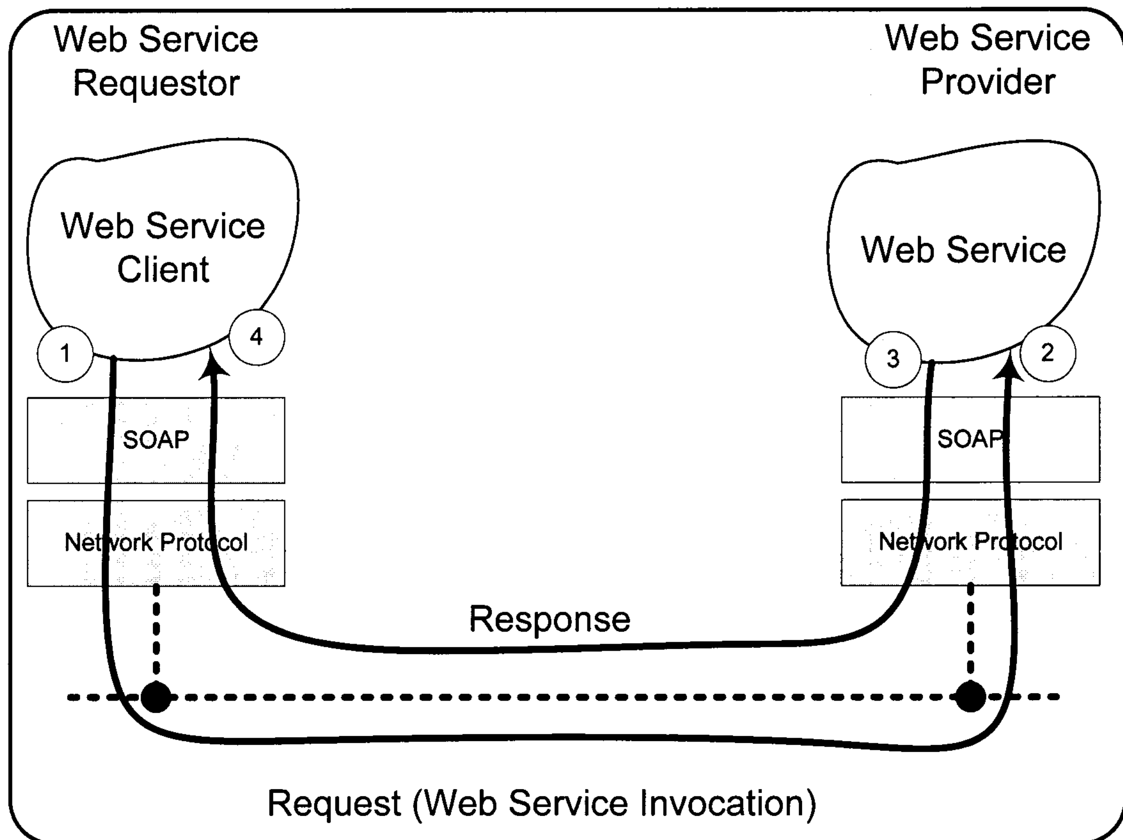

Figure 2.4 XML-based messaging in SOA

2. When this request is received by the Web Service provider's infrastructure, the SOAP message (the request) is forwarded by the network to the Web Service provider's SOAP runtime (for example, a SOAP server) who routes it to the Web Service. Eventually, a conversion into programming language-specific objects is performed based on the conversion schemes contained within the message.

3. The Web Service processes the request and builds a response, which is also a SOAP message. The SOAP infrastructure of the provider sends this response to the requestor over the network.
4. The reception of the response by the requestor is the last step. The network infrastructure routes the message through the SOAP. The response message is then presented to the requestor's application.

2.4 Management areas

For the convenience of standardization, management activities in distributed systems are subdivided into five functional areas [23]: Fault management, Configuration management, Accounting management, Performance management, and Security management. The acronym FCAPS is usually used to refer to these functional areas.

2.4.1 Fault management

A fault is triggered when a component of the system behaves incorrectly. A fault can be either persistent or transient and measures taken to correct this fault depend largely on this property. While logging for eventual analysis is suitable for most transient faults, persistent faults may have to be corrected. Fault management includes fault detection, fault location, fault isolation, and fault repair. Fault detection is the first step in fault management and consists of concluding that misbehavior occurred in one (or more) component(s). Once a fault is detected, the faulty component should be identified and located during fault location. The sooner a faulty component is isolated, the less will be the related cost. If the fault is considered as fatal, fault isolation aims to remove the faulty component in such a way that the fault

will not propagate to non-faulty components. The information gathered from previous steps can be used later during fault repair.

2.4.2 Configuration management

Configuration management is used to collect information on available resources. All the resources, including the failed ones, are listed and their details are stored and kept up-to-date. This information is collected on a regular basis or whenever a change is made.

2.4.3 Accounting management

Accounting management deals with the usage of available resources and probably the billing. Relevant users and managers are informed on a regular basis of the amount of resources they are using. Accounting management is very crucial if the available resources are limited with regards to potential users.

2.4.4 Performance management

Performance management consists of gathering information and statistics on system performance during both heavy and low system load. The resources that affect the performance of the system should provide mechanisms for data collection, both solicited and unsolicited. Response time and throughput are basic criteria for performance management.

2.4.5 Security management

Security management deals with traditional security issues: legitimate use, confidentiality, and data integrity. Based on the access rights defined for different resources, it should enable or

disable the access to system functionalities especially those functionalities used for system control. It should continuously monitor the system for security rules violations and take corrective action whenever needed.

Misbehavior detection is a quite hard task. Testing is a widely used mechanism to perform this task. In the next section, I present testing activities and show the difference between active and passive testers.

2.5 Testing for fault detection

Detecting misbehaviors during the development of a Web Service is mainly based on applying certain number of carefully generated test cases. A tester applies these test cases to the Web Service Under Test (WSUT) and checks its responses. If the received response is different from what is expected, then a fault has occurred. This process is known as *active testing* and is usually conducted before deployment or whenever a user reports a fault.

If a Web Service is already deployed, another kind of testing, known as *passive testing*, is more suitable. Passive testing consists of the observation of messages exchanged between a Web Service and its environment and the verification of the correctness of these messages. Compared to active testing, the input is not assumed to be correct (or non faulty). The main component in a passive testing system is the *passive tester* known also as *observer*. The number of passive testers and their locations are a key issue for fault location, fault isolation, and fault repair in passive testing activities.

2.5.1 Active testing

Active testing refers to the process of applying a set of inputs to an Implementation Under Test (IUT) and the verification of its reactions. In this configuration [24], the tester has complete control over the inputs and uses selected test sequences to reveal possible faults in the tested system. Active testing comprises three main tasks: test cases extraction, test cases execution, and result analysis.

2.5.1.1 Test cases extraction

In this first step, *optimal* test cases are extracted from the initial requirements of the IUT. An optimal test case is one that provides high fault coverage while generating the lowest cost. Many generation methods have been studied ([24], [25], [26], [27], [28]), and most of them are based on formal methods ([29], [30]). The efficiency of a generation method is basically measured with regards to its fault coverage.

2.5.1.2 Test cases execution

This step consists of the application of the test cases resulted from the previous step to the IUT. A testing architecture should be used for this purpose. ISO 9646 ([31]) defines four types of test architectures for conformance testing: local architecture, distributed architecture, coordinated architecture, and remote architecture (Figure 2.5). The architectures differ in their capacity of observation which affects its fault detection capabilities. From the configuration point of view, they differ in the arrangement and communication between the Upper Tester (UT), Lower Tester (LT), IUT, and the Point of Control and Observation (PCO). PCO are the points where testers have access to the IUT.

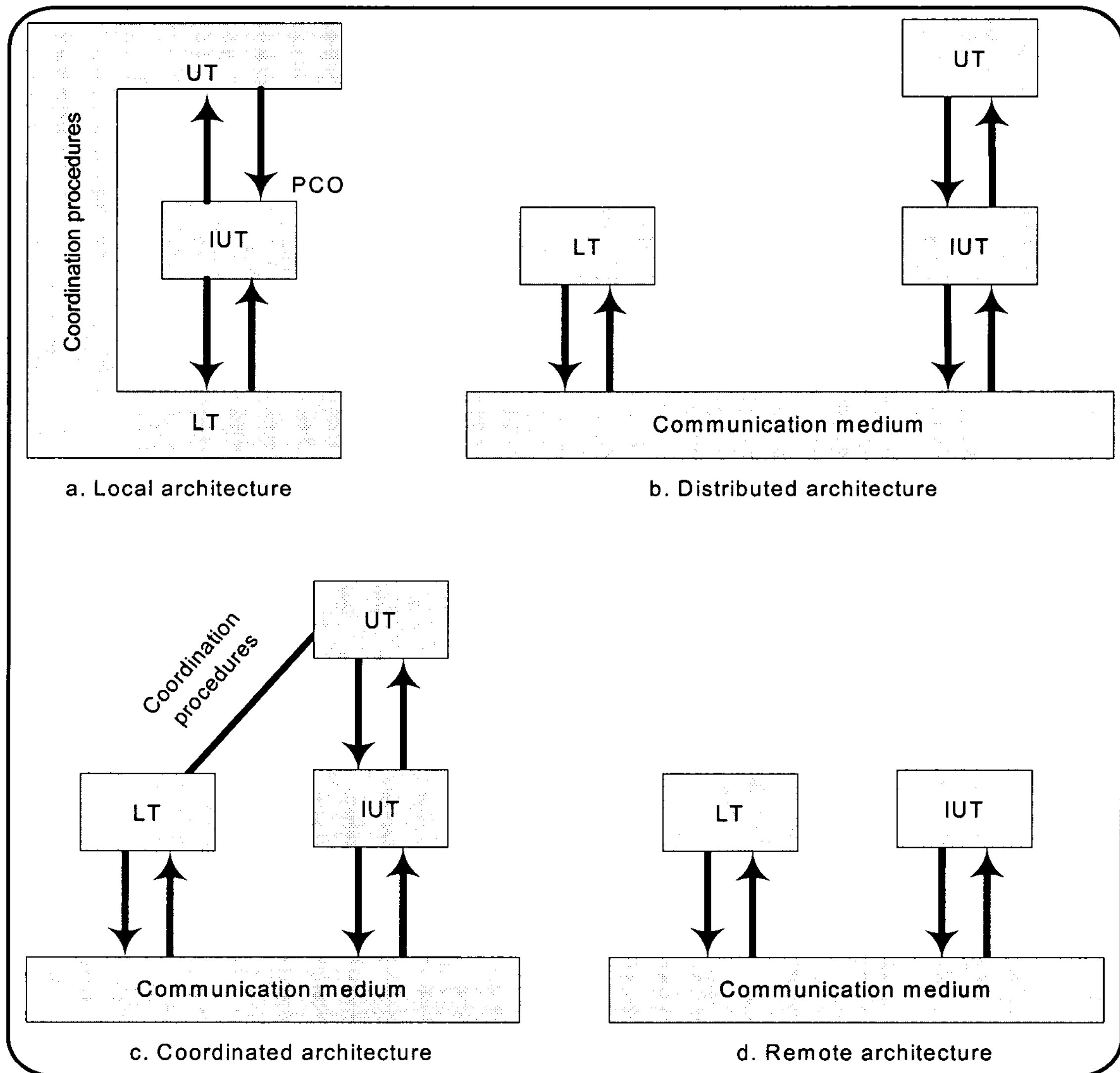

Figure 2.5 Standard active testing architectures

2.5.1.1 Results and analysis

During this step, observed responses (from the IUT) and the expected ones are compared and a verdict is issued. If the expected and observed responses are similar, the verdict is "Pass". If they are different, the verdict is "Fail"; "Inconclusive" if no final verdict can be reached.

2.5.2 Passive testing

Even though active testing is performed on all systems before deployment, it is not practical once a system is operating in its final environment. Under normal conditions, we have no control over the inputs and outputs exchanged between components of a system and can only observe these interactions *passively* (Figure 2.6). This passive observation is the basis of passive testing and is performed by an observer. In fact, an observer observes the input/output (behavior) of a component during normal operations for the purpose of detecting misbehaviors without disturbing the System Under Observation (SUO). Disturbing here should mean no injection of input/output messages for testing purposes. If the observed input/output is different from what's expected, the component is then declared faulty.

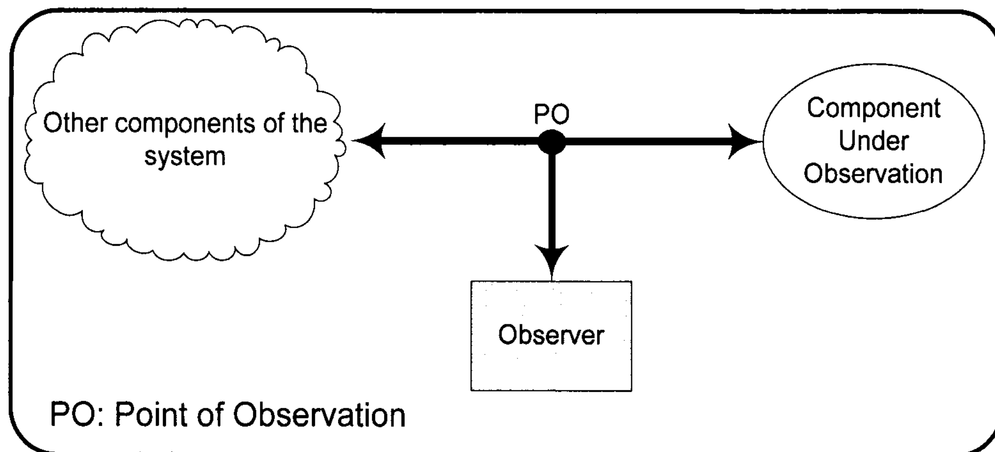

Figure 2.6 Passive testing architecture

In order to be able to conclude that a fault has occurred, the observer must have enough knowledge on how the components being observed should react to a set of visible events.

This set of events depends on what events should be inspected. Comparing observed behaviors requires one of the following two concepts [32]:

- *Redundancy*: In this approach, there are several (redundant) copies of the SUO and the detection relies on the assertion that a fault occurring in one copy is unlikely to be observed at the same time in other copies. The observer compares the reaction of all the copies and declares faulty the component that reacts differently.
- *Reference*: the behavior of the component being observed is known, defined, and available to the observer before the observation starts.

Redundant copies approach has several major drawbacks. The most inconvenience is that all these copies should be developed by different teams and must be running in parallel requiring different hosting systems or degrading the performance if running on the same system. The reference approach seems to be more efficient but a new problem arises: how to represent the required knowledge?

There are two interesting ways to design an observer based on the reference approach: *expert systems* or *model-based* [33]. Expert systems depend on human experience and are applicable only to systems that have been encountered previously. Model-based observers use the reference model of a system and can detect all faults expressed in this model. The number and types of faults detected by the latter approach depend on the expressiveness of the used model.

The model-based passive testing procedure is generally divided into two steps:

1. *Passive homing* (or *state recognition*): in this step, the observer's specification (the model of the observer) is brought to a state equivalent to the one that the

implementation's specification (the model of the implementation) might be in. If no such state is found, a fault is immediately reported. The set of inputs leading to this state is known as the homing sequence.

2. *Fault detection*: starting from the state identified in the previous step, the observer checks the observed behavior against the system's specification. If an observed event is not expected then a fault is immediately reported.

FSM, Communicating Finite State Machines (CFSM), and EFSM have been used as models for passive observation. The next paragraphs will define these models and the associated faults models.

2.5.2.1 FSM

A Finite State Machine M is defined as a tuple $(S, s_0, X, Y, D_S, \delta, \lambda)$ [24], where:

- S is a set of states,
- $s_0 \in S$ is the initial state,
- X is a finite set of inputs,
- Y is a finite set of outputs,
- $D_S \subseteq S \times X$ is the specification domain,
- $\delta: D_S \rightarrow S$ is the transfer function, and
- $\lambda: D_S \rightarrow Y$ is the output function

The machine starts at s_0. Whenever an input is received, λ computes the corresponding output and δ returns the corresponding next state(s).

The fault model associated to an FSM is related to the transfer function and the output function in such a way that the following faults can be detected:

- Output fault: a transition has an output fault if, for the corresponding state and input, it produces an output different from what's computed by λ
- Transfer fault: a transition has a transfer fault if for the corresponding state and input, the next state is different from what's returned by δ
- Transfer fault with additional states: this fault is similar to the previous fault except that the next state does not exist in S
- Additional or missing transitions: this fault occurs when extra transitions are added to the initial machine.

2.5.2.2 CFSM

A communicating FSM consists of a set of FSM machines (M) that communicate through a set of channels (C). A CFSM N can be denoted by N = (M, C) [34] where:

- M = $\{m_1, \ldots, m_n\}$ is a set of n machines
- C = $\{C_{ij} \,/\, i,j \leq n,\ i \neq j\}$ is a set of channels, where each element C_{ij} denotes a communication channel from m_i to m_j. It behaves like a First In First Out (FIFO) queue with m_j taking inputs from the head of the queue and m_i placing output to the tail of this queue for messages generated by m_i and intended to m_j.

Starting from its initial state, a machine can consume an input that is in its associated channel or produce an output and put it at the queue of another machine's channel.

Modeling a system as a CFSM allows the detection of numerous faults. Deadlocks, livelocks, and unspecified receptions are examples. Faults related to behaviors of channels (loss, duplication, overflow ...) can also be detected.

2.5.2.3 EFSM

EFSM is an extension of FSM models by the following:

- Interactions have certain parameters, which are typed.
- The machine has a certain number of local variables, which are typed.
- Each transition is associated with an enabling predicate. The predicate can be any expression that evaluates to a Boolean (TRUE or FALSE). It depends on parameters of the received input and/or the current values of the local variable.
- Whenever a transition is fired, local variables can be updated accordingly and parameters of the output are computed.

Formally, an EFSM is described by a tuple M = (S, s_0, I, O, T, V) ([24]) where:

- S is a set of states,
- $s_0 \in S$ is the initial state,
- I is a finite set of parameterized inputs,
- O is a finite set of parameterized outputs,
- T is a finite set of transitions
- $\delta: S \times (I \cup O) \rightarrow S$ is a transition relation

In an EFSM, each transition of T can be represented as t: $S_s | I | P | A \mid O | S_e$ where:

- t: label/ID of the transition,
- S_s: starting state of the transition,

- I: the input that triggers the transition,
- P: the enabling predicate (data conditions),
- A: variables assignments
- O: the output produced by the transition
- S_e: ending state of the transition

From state s_0, the machine fires transitions as it receives inputs and its predicate is evaluated to TRUE. When firing a transition, local variables are updated, parameters of output message are computed, and the next state is identified.

The faults that can be detected when using EFSM models include wrong initial values, wrong specification of data type, wrong specification of data representation, referencing an undefined variable or a wrong variable, and arithmetic and manipulative faults.

After giving an overview of the most used models in passive testing, the next section will illustrate, through an example, how faults can be detected where the behaviour of the WSUO is specified as an FSM machine.

2.6 Example of fault detection by passive observation

Let's consider the example depicted in Figure 2.7 where S and I are the FSM machines of the specification and an implementation of a system, respectively. If I has an observed behaviour that is different from that of S, then I is *faulty*. Formally, this can be expressed as:

- if we observe x/y, an I/O sequence of I from a state s (i.e. $\exists s \in S_I / \lambda_I(s,x) = y$)
- if $\neg\exists s' \in S_S / \lambda_S(s',x) = y$ then I is faulty

In Figure 2.7, the dashed line represents a fault in the implementation. If both the observer (using machine *S*) and the implementation (machine *I*) are in state *S2*, the generated output for *0* as input is *0*. While this is correct for both machines, reception of another input *0* will lead to a fault since *S* will expect *0* as output and *I* will produce 1. The specification *S* is assumed to be correct and discrepancies between the expected and the observed behaviours are due to a faulty implementation.

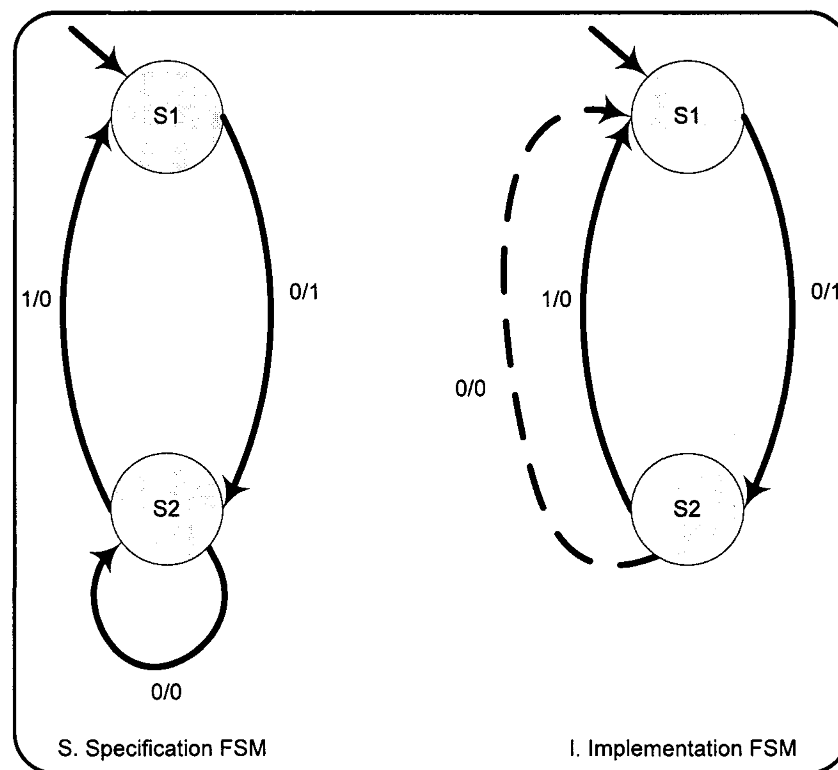

Figure 2.7 Specification and implementation FSM machines

The observer takes as input the model of the component to be observed, compares the input/output sequence to the sequence expressed in the model. In case of discrepancy, a fault is detected. If no fault was detected within a certain interval of time, the observer con-

cludes that the component is fault-free with regards to the set of observed events. The observer can be either *online* or *offline*: an online observer checks in real time the exchanged input/output while an offline observer uses the log files that might be generated by the observed component itself. Unlike the offline observer, an online observer has the advantage of detecting faults when they occur but has the disadvantage of being more complex and resource demanding.

Active testing provides a better fault coverage if an efficient generation method is used since *optimal* test cases are used and most of possible scenarios will be investigated. The test cases generation process is also the main weakness of active testing. In fact, the use of a generation method requires additional overhead and skills. Moreover, the execution of the test cases may affect considerably the performance. In passive testing, no test cases are generated and testing activities can be completely transparent to the implementation under test. The main weakness with this kind of testing is that there is no guarantee on fault coverage during observation periods.

2.7 Summary

This chapter gave background information on concepts that will be used in upcoming chapters. I first introduced the Service-Oriented Architecture, its roles and operations, and its programming stack. Section 2.3 presented an example of a Web Service and different steps to invoke it. Since Web Services are being more and more used and standardized in many IT sectors, their management is becoming crucial for their success. I presented different management areas that have to be considered while developing management solutions of Web

Services. I also showed how testing, both active and passive, can be used for misbehavior's detection and gave an example of fault detection with passive testing.

In the next chapter, I will present state of the art of the concepts presented in this chapter. This will include management of Web Services (both functional and non-functional) and passive testing which will be used as the basis for my contributions to management of Web Services.

Chapter 3
Related Work

The purpose of analysis is not to compel belief but rather to suggest doubt - Imre Lakatos

The contributions of this thesis are related to management of Web Services namely functional and QoS management using passive observation. The next section discusses the state of the art on management of Web Services. This will include proposals from both academia and involved industries. In section 3.2, related works on passive testing are presented and their limitations are highlighted.

3.1 Web Services management

Most of the work on Web Services focus on their development and deployment. Management of Web Services [35], and in particular, fault and performance management, is not yet a well-studied area. However, some interesting works have to be cited.

Existing approaches for management of Web Services include approaches from network management and those that have been developed specifically for Web Services. The approaches that have been used for network management for a long time seem to be a candidate for the management of Web Services. However, their main drawbacks are due to the major differences between Web Services and network components, and the need for the participation of a component in its management. In fact, most network components (devices) run standardized protocols that have specific and known attributes to be managed. If run-

ning proprietary/non-standard protocols and/or applications, the manufacturer of the component usually provides specific management agents/applications or well defined sets of APIs.

In network oriented approaches, Simple Network Management Protocol (SNMP) ([36]) is based on TCP/IP and the client/server communication mode. An agent, associated to a Management Information Base (MIB) [37], communicates with a management station by processing *get* (send the value of an attribute) and *set* (modify the value of an attribute) messages and generating *trap* messages (unsolicited notifications). Thus, SNMP management system requires an SNMP agent, a MIB, and a management station (manager).

Common Management Information Protocol (CMIP) [38] fulfills in the OSI reference model protocol stack [21] a role similar to that of SNMP in TCP/IP. CMIP has many advantages compared to SNMP including the number of available commands and the possibility to operate over TCP/IP. However, complexity and long development time, especially CMIP Over TCP/IP (CMOT) [39], have kept its adoption pervasively.

Web Services community is still trying to determine the requirements and define specific approaches for Web Services management. These approaches can be divided into two main groups: approaches based on active testing and approaches requiring the Web Service (architecture) to support management interfaces. The World Wide Web Consortium presents some of the requirements that Web Services management architectures should satisfy to provide management features [40]. It includes the definition of standard metrics, management operations, and methodologies for accessing management capabilities. The expected architecture must provide a manageable, accountable and organized environment for Web

Services operations. At least, resource accounting, usage auditing and tracking, performance monitoring, availability, configuration, control, security auditing and administration, and service level agreements should be supported. An approach where the Web Service provides specific interfaces for management is presented in [41]. The developer is supposed to supply commands and APIs for operations that are invoked by the management system.

In [42], the authors classify the management of Web Services into three levels: infrastructure-level, application-level and business-level. The infrastructure-level deals with the Web Service platform while the application-level focuses on the Web Services themselves. The business-level takes into consideration the conversations between a Web Service and its client; a framework and a tool (collectively known as Web Service Manager) to perform management at this level have been developed.

Management approaches presented in [40], [41], and [42] suppose that the Web Service will provide management operations that one can invoke. Developers of Web Services have to develop and deploy these operations in addition to the business operations the Web Service is offering.

A couple of management tools to be integrated into Web Services' environments are already available. Hewlett Packard's Web Service Management Engine ([43]) is a collection of software components that enables some management features including the definition and the enforcement of Service Level Agreement (SLA). Parasoft ([4]) provides a set of tools (SOAPTest, .TEST, WebKing) to assist during the life cycle of a Web Service. SOAPTest, for example, helps preventing errors by performing server functional testing, load testing, and client testing. It parses the WSDL of the target Web Service and generates test suites to

test each of the exposed operations. These tools have to be installed and configured, thus requiring extra resources and introducing new cost for Web Services providers.

There have been a considerable amount of works on testing of Web Services in the last couple of years. They can be divided into two main groups: works targeting functional aspects of Web Services, and works tackling non-functional aspects. The first group deals with the correctness of interactions between Web Services and their clients. The second group is concerned with performance and Quality of Service of Web Services management. Related work on both groups is presented in the following two sub-sections.

3.1.1 Functional management

For functional management, the majority of works is based on active testing. For this kind of testers and as introduced in section 2.5.1, appropriate test cases have to be carefully generated, executed, and their results analyzed. Even if this task is mandatory, there are limitations to active testing. First of all, exhaustive testing is impractical for quite large Web Services. In fact, test cases can not cover all possible execution scenarios that a Web Service will have to handle while serving clients' requests. The size of test cases is bounded by the cost a Web Service provider is willing to spend on testing activities. Usually, active testing stops whenever developers are confident that the Web Service is good enough to be put into the market.

Few works describing test cases generation methods for Web Services have been published recently; most of them are based mainly on static analysis of WSDL documents. Xiaoying et al. [44] present a method for test data generation and test operation generation

based on three types of dependencies: input, output, and input/output. In [45], the authors propose a method for test data generation. A set of tests is randomly generated based on the WSDL document. Mutation techniques are then used to improve the quality of the test suite. Mutations based on WSDL documents are also presented in [46]. In [47], the authors combine both EFSM models and WSDL documents to generate test cases. For composite Web Services, methods for generating test cases are proposed in [48] and [49]. In the first paper, the authors use WSDL documents, Task Precedence Graph (TPG), and Timed Labeled Transition Systems (TLTS), while in the second model, a model checking tool (SPIN) is used.

Approaches in [44], [45], [46], [47], [48], and [49] are based on active testing and thus cannot be used for management of already deployed Web Services.

3.1.2 QoWS management

Quality of Web Services (QoWS) management includes definition of QoWS attributes, QoWS publication, discovery, validation, and monitoring. Existing approaches for QoWS management can be classified into two groups:

1. Extending related technologies including WSDL and UDDI to support QoWS.
2. Mandating independent entities to perform some or all of QoWS management tasks.

In the first category, [50] extends SOAP header to include QoWS information. WSDL is also extended to describe QoWS parameters, their associated values, computation units (e.g. millisecond, request/second), etc. UDDI extension consists of extending the current UDDI

data structure with QoWS information [51]. The aim of these extensions is to allow QoWS-based publication and discovery of Web Services.

In the second group, works present solutions for one or more of the following QoWS management operations:

- QoWS attributes' definition: the first step in QoWS management is the definition of evaluation's criteria and attributes. A set of attributes have been defined, studied and used in software engineering for a long time ([52], [53], [54]). However, the dynamic nature (e.g. composition) of most of Web Services requires consideration of new attributes such as availability, thrust, and reputation ([55], [56]).
- QoWS publication and discovery ([57], [58], [59]): this operation allows providers to include QoWS information in WSDL. This information is then used by requestors when finding Web Services to select the appropriate Web Service in terms of functional and QoWS requirements.
- QoWS verification ([58], [59], [60]): this operation allows the provider to certify that the QoWS claimed by the Web Service is accurate before invocation and during interactions.
- QoWS negotiation [58]: if the available published QoWS requirement do not satisfy client's needs, negotiation operations' strategies can be followed to reach an agreement on different QoWS attributes.
- QoWS monitoring ([61], [62], [63],): performs monitoring of the Web Service during interactions with clients to assess if the QoWS attributes agreed upon in previous points are delivered.

3.1.3 Discussion

All the solutions presented above fit in one or more of the following categories:

1. Platform-dependent
2. Assume that a Web Service will participate in its management by providing specific interfaces (e.g. W3C architecture),
3. Are based on active testers.

Utilization of platform-dependent management approaches is restricted to the targeted platform. When management features are embedded to the hosting platform, they are only available to the provider and cannot be used by clients or third party certification entities. The client is forced to rely on management information made available by the Web Service's provider and has no mean of verifying it. Moreover, information used in assessing the behavior is taken from one location: at the provider's side. There are many situations, in composite Web Service for example, where this information should be gathered from different sources and locations. This requires an additional property of management solutions: interoperability between heterogeneous involved management entities.

The Web Services architecture becomes more complex if it has to support management aspects in addition to its basic functions. The performance of the Web Service and its hosting platform is also degraded due to these additional features. Moreover, developers of Web Services have to implement also the needed interfaces and APIs to support management. Since these features will be used somehow sporadically, the return on investment of their development and deployment might be relatively low.

Once a Web Service is deployed, active testing cannot be used to test, on the fly, the correctness of interactions of the Web Service with its clients. Moreover, application of generated test cases consumes resources and may disturb the Web Service.

Since management of Web Services is somehow at its earlier stages, related work usually concentrates more on provision of management features without evaluating the overhead they generate. In order to select the appropriate management approaches, a potential user must be able to assess it in terms of usefulness and associated cost.

To solve some of the limitations of related work cited in this section, I propose novel architectures for management of Web Services. These architectures are rooted in passive testing to perform online and transparent observation of Web Services.

While the above section gave a literature review of management of Web Services, the next section will provide an overview of related work on passive testing; a concept that will constitute the core of the contributions of this thesis as will be shown in forthcoming chapters (Chapter 4 to Chapter 8).

3.2 Passive testing

Many models have been used for model-based observers but most of the published works on passive testing are on control part of systems and use the FSM model. This model has been specially used for fault management in networks. The authors in [64] present techniques and algorithms for fault detection for network management based on FSM.

In [65], an approach based on causality invariants expressing the set of critical properties of the system is proposed. An invariant expresses that each time the SUO performs a given sequence of input/output actions, it must show behaviour reflected in the invariant. The in-

variants of a system can be explicitly (manually) extracted from an FSM specification. The authors in [65] use a pattern matching algorithm that supposes that the whole trace is available and thus there is no need for the homing procedure.

In previously cited works, the whole network is represented by a single machine with inputs and outputs on transitions. This machine might be huge even for a relatively small network due to the *state explosion* problem. Moreover, fault location or fault isolation is difficult if only one FSM is used to specify the whole network.

In an attempt to resolve some of these problems, Miller et al. proposed in [66] a CFSM specification of network components. Fault location and isolation are enhanced in this case if observers are placed in appropriate locations ([67], [68], [69]). Even though this model is more realistic than single FSM, it considers only the control part of observed systems with no consideration of the data flows.

Data flow using EFSM has been studied in trace analysis ([70], [71], [72], [73]) and diagnosis ([74], [75], [76], [77]) where the authors assume there exist a complete trace. Consequently, there is no need for the passive homing procedure. In some cases, the analyzer in diagnosis and trace analysis is part of an active tester. In other cases, the whole traces are handed to the traces' analyzer.

When using FSM models with incomplete traces as input, the homing procedure is necessary and consists of finding the state where the implementation is actually located. The state is in fact a label. The usage of EFSM adds more complexity to the homing procedure since variables have to be correctly initialized in addition to state's label recognition. If the

EFSM is extended with timing properties, the clocks should be initialized with the correct values before going forward to the fault detection.

Very few works consider the homing procedure in EFSM-based passive testers. In [78] and [79], the concept of invariants studied in [65] is extended to support EFSM. However, there is no consideration of homing procedure since it addresses a complete trace. Even for fault detection, the authors do not verify if the values of variables are correct, but rather check if they are compatible with the data type signatures of corresponding parameterized messages.

In ([80], [81]), an event-driven EFSM approach is introduced to deal with variables initialization in passive testing. An event is either an input to SUO or an output from it, but not both. Variables are initialized using information from the predicates on transitions and relations between variables.

The homing procedure in an EFSM-based observer consists of recognizing the actual state of the SUO in addition to assigning appropriate values to different variables. At start-up of homing, all states are possible and variables can have any value in their definition domains. The objective is to determine the actual state's label and a unique value for each variable.

In the few published papers on EFSM-based passive testing, the homing procedure is either ignored or it depends fully on the upcoming observed input/output traces. In the first case ([78], [79], [82], [83]), the observer must get all the traces to be able to initiate the fault detection process. In the second case ([80], [81]), the observer waits for exchanged messages before moving forward in the homing procedure.

Ignoring the homing phase is based on the assumption that the passive tester has access to a complete trace. When the observer has access to an incomplete trace, it cannot assess the correctness of the trace.

Even if the homing procedure is considered, fault detection may be delayed (even forever) if the observer waits for exchanged messages to continue on the homing procedure. Moreover, if there is a significant time gap between inputs and their corresponding outputs, the observer spends most of its time waiting.

Observers can benefit from this time gap to find pertinent information in the EFSM model. In fact, analysis of possible previous paths can help reduce the number of possible states and possible values of variables as will be proposed and demonstrated in Chapter 8.

3.3 Summary

This chapter presented the state of the art of management of Web Services and passive testing. It started by discussing proposed management approaches including those derived from network management. Management is then divided into two categories: functional and non-functional. Functional management relates to the correctness of request/responses in terms of their occurrences and contents. Non-functional management deals mainly with QoWS. Related works of both categories have been presented and their limitations have been highlighted.

The chapter also presented and discussed related work on passive testing. Most of previous works in passive observation consider FSM models which do not support data flow. The approaches supporting data flow using EFSM models are not always inefficient. The majority of these works supposes complete traces and do not consider the homing procedure.

Works where the homing procedure is considered are based on observed events which may delay fault detection especially if time gap between events (requests and responses) is somehow high.

In the next chapter, I will present core ideas of my contributions to management of Web Services. Contributions presented in following chapters will be based on extension and development of these ideas.

Chapter 4

Extending SOA with Observation Capabilities

If you want to succeed, double your failure rate -
Thomas J. Watson

This chapter presents an extension to the Service Oriented Architecture with observation capabilities. The objective is to design a new management approach with a set of interesting properties. It is highly desirable that the management architecture can be made available to Web Services' providers, their clients, and third party entities depending on who is interested in management of a specific Web Service. A Web Service's provider can use the architecture to make sure that the Web Service she/he is offering is operating correctly with regards to both functional and non-functional (such as QoWS) aspects. The client can make use of the architecture in order to verify that the Web Service she/he is using is operating as agreed on with the Web Service's provider. Furthermore, a third party mandated by the client and/or the provider, can use the architecture to provide an independent certification stating if the Web Service is conforming to its specification.

The transparency of online management activities is of prime importance during Web Service offering. Neither the Web Service nor the client should notice that such activities are taking place. This implies that no additional messages related to management purposes will be sent and/or received to/from the client/Web Service. The network load generated by

* Results from this chapter have been published in [84].

management activities should also be minimized to keep the service performance at an acceptable level.

Online detection is another important property of management architecture. The design of the architecture should allow detection of misbehaviors as soon as they appear or within a reasonable delay.

In the next section, we will develop further the key ideas of the proposed approach for management of Web Services and the extension of the SOA with respect to the requirements discussed above.

4.1 Extended SOA

To fulfill the requirements discussed in the previous section, mainly the online property, passive observation is the proposed approach. Moreover, the SOA is extended with observation capabilities by developing a specific Web Service observer. The proposed architecture is based on two concepts namely *passive testing* and *Web Services*. Detecting misbehaviors without application of specific test cases necessitates the use of passive testers (observers) (section 2.5.2). Moreover, making this observer available to providers, clients, and third parties can easily be achieved by designing the observer itself using the same paradigm: Web Services. These two design concepts give the architecture a set of interesting features to perform management of Web Services as it will be presented in the next sub-section.

4.1.1 ESOA Features

The following are interesting features that characterize the proposed architecture:

- **Online detection:** the observer should be able to detect misbehaviors as soon as they are generated by the observed Web Service. This implies that the observer(s) must receive, using appropriate mechanisms, all messages exchanged between the client and the Web Service. A thorough study of possible mechanisms to collect these traces will be presented in section 6.1.
- **Minimal overhead:** even if management activities will induce some extra load (extra packets in the network, performance degradation, etc), the idea within the proposed approach is to keep this load as low as possible. Consequently, the designed observer should never generate messages just for the sake of management except when reporting the result of the observation.
- **Transparency:** in a completely distributed architecture, the design and the implementation of the observer(s) should guarantee the transparency property. Neither the Web Service nor the client should notice that management activities are taking place, and the overhead will not require abusive additional resources at the client and/or the Web Service's side.
- **Platform-independency**: interacting with a Web Service is always independent from its hosting platform. The design of the observer in the proposed approach as a Web Service will allow this independency. The interaction with this observer is not tied to its platform, does not require additional software, and needs only limited knowledge and resources.

4.1.2 Procedure and Components

Management of Web Services using the proposed architecture can be performed following the SOA paradigm and observation functionalities. The steps are listed hereafter and illustrated in Figure 4.1

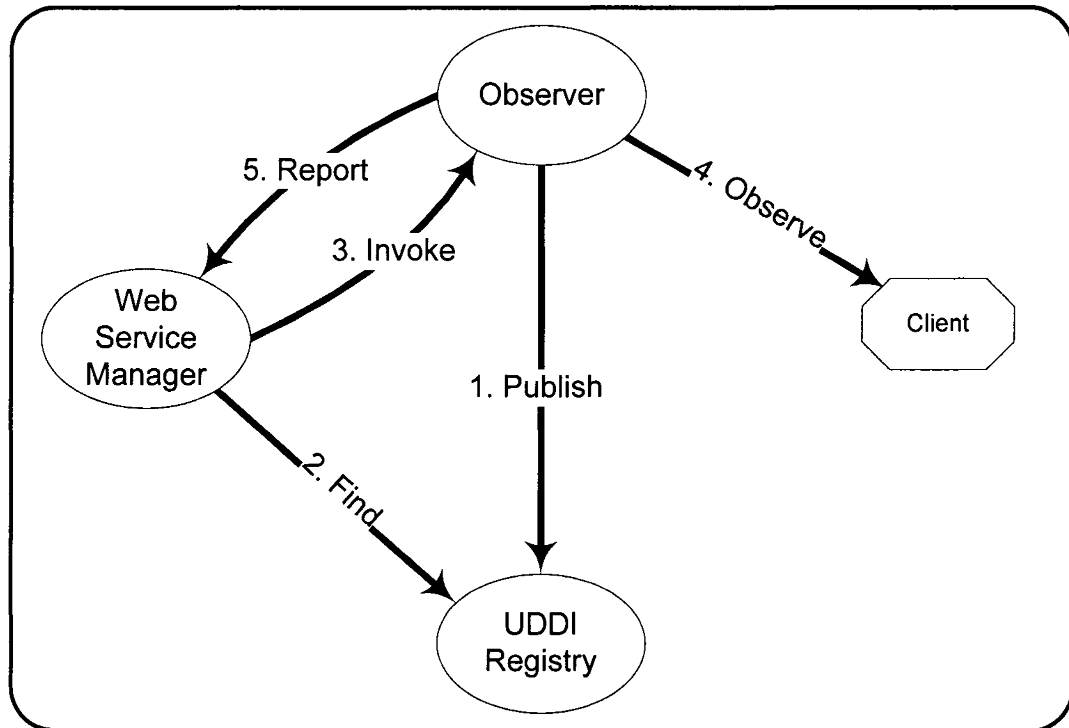

Figure 4.1 ESOA with passive observation

1. **Publish**: once the Web Service Observer (WSO) is developed and deployed, its WSDL document is then published. This can be done by Email, FTP, HTTP, UDDI, etc. Figure 4.1 shows publication using UDDI registries.
2. **Find**: this action depends on how the publication process had been performed. It can consist of opening an email inbox, connecting to a FTP server, issuing a get request to HTTP server, or a request to a UDDI registry.

3. **Invoke:** The WSO is first invoked by a manager. The manager can be the client, the provider of the Web Service, or an authorized third party. In all cases, the manager should provide specific information during invocation. This information includes the location of the Web Service to be observed, specification of the expected behavior (model), when to start observation, when to end observation, etc.
4. **Observe:** Once the WSO is invoked, it waits for exchanged messages between the Web Service and the client. The observation starts and ends at the time specified during invocation of the WSO.
5. **Report:** When the observer stops observation (misbehavior detected or observation period expired), it returns the result to the manager. Based on these results, fault location, isolation and correction processes are initiated if a fault occurred.

The architecture requires a set of information/resources to be able to passively observe a Web Service. These requirements are discussed in the next sub-section.

4.1.3 Number of observers and location of points of observation

The observation in distributed architectures requires the selection of a number of observers and the best locations for points of observation where traces are collected. The number of observers and location of the points of observation affect significantly the detection capabilities of the architecture. For example, if the WSUO is a composite Web Service, it might be more interesting (in terms of misbehavior's detection) to consider a network of observers: an observer for each Web Service rather than a unique observer for the composite Web Service. In such architecture, the cooperation of all observers can generate pertinent information for

Web Services management. The consideration of a global observer (for the composite Web Service) and local observers (for composing Web Services) presents a framework where this cooperation can be orchestrated for the benefit of better fault detection.

ESOA allows two different architectures: a mono-observer architecture and a multi-observer architecture. The first architecture, depicted in Figure 4.2, is useful when the WSUO is a simple (non-composite) Web Service. Traces collected by observing the interactions between WSUO and its client are sufficient enough to analyze and decide on the correctness of the observed behavior.

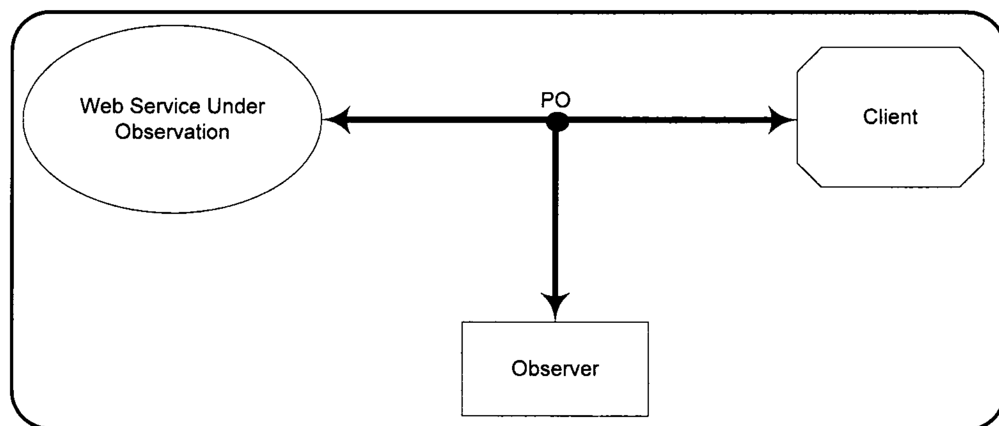

Figure 4.2. Mono-observer architecture

When the WSUO is a composite Web Service, the mono-observer architecture might not provide enough information for management. In fact, a misbehavior detected within the composite Web Service can originate from basic Web Services. Furthermore, some faults may occur due to the composition itself, these faults are known as feature interaction. Figure 4.3 shows multi-observer architecture where a composite Web Service is being observed as well as its two basic Web Services. The global observer and the two local observers cooper-

ate in detecting and locating misbehaviors. This architecture can also be used for mass usage testing where many clients are invoking the same Web Services to assess, for example, its scalability. However, in this thesis, the multi-observer architecture is mainly used for composite Web Services.

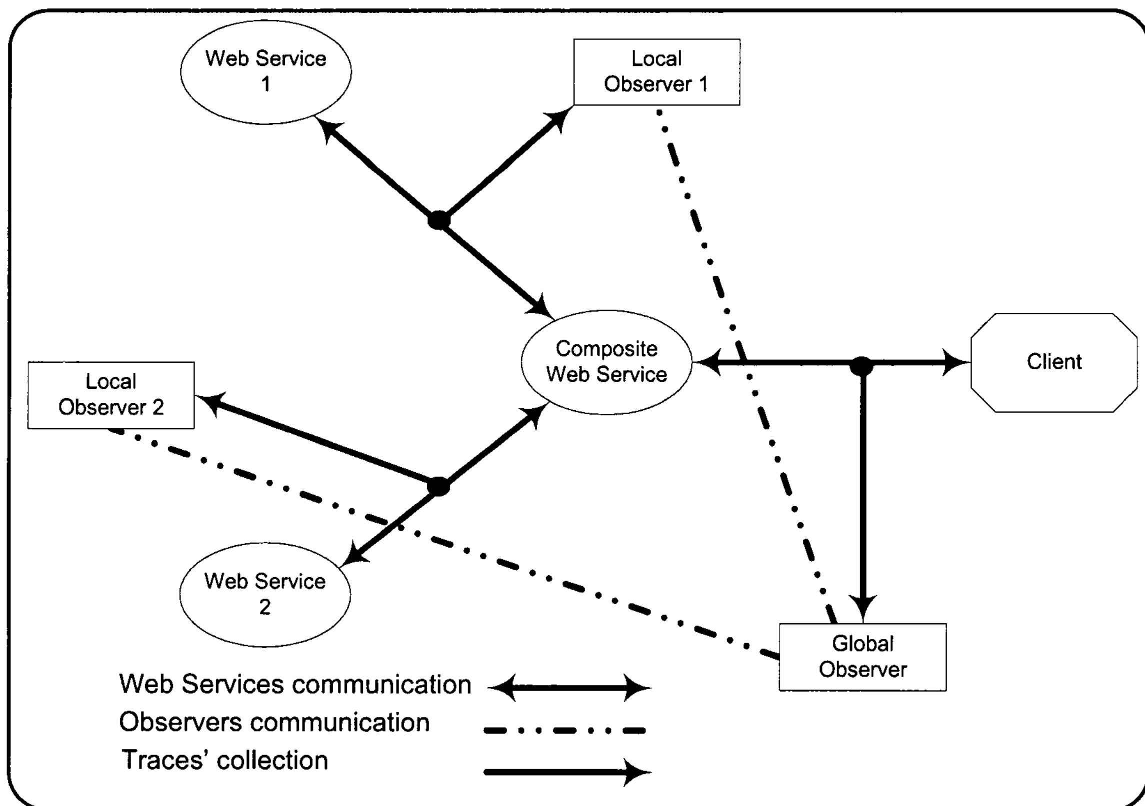

Figure 4.3. Multi-observer architecture

Three types of interactions are illustrated in Figure 4.3. Web Services' communication refers to the SOAP-based communication between Web Services and their clients. Traces' collection consists of forwarding messages exchanged between a WSUO and its client to local observers. Observers' communication conveys information between observers. The latter information is divided into three categories:

1. Configuration information: during configuration of different observers, local observers must indicate to the global observer which Web Service they are observing and where they are located. The global observer needs this information to identify observers and associate the traces it will receive to appropriate observer/WSUO.
2. Traces from local observers to the global observer: whenever a local observer gets a trace, it sends it to the global observer.
3. Notifications of faults: if the global observer detects a fault, it informs other local observers. In the case where a local observer detects a fault, it informs the global observer. The latter informs remaining local observers that misbehavior has been observed elsewhere and they should be aware of some specific traffic/actions.

4.1.4 Requirements

In order to observe a Web Service, the architecture bases its misbehavior's detection on two types of information: *specification of the expected behavior* and *exchanged traces*:

- Once the observation starts, all exchanged traces between WSUO and its client must be forwarded to observers. Messages exchanged before initiating the observation are not available to the observers, which necessitate the use of a homing procedure.
- The observer must have unambiguous information about the expected behavior of the WSUO. This information should cover both functional and non-functional (QoWS) behaviors.

In the next chapter, I will discuss different ways to represent expected behaviors of WSUO and present in details the ones I will be using in the remaining chapters of this thesis. Possible traces' collection mechanisms are discussed in Chapter 6.

4.2 Summary

This chapter presented an extension to the SOA to allow management of Web Services using Web Services-based passive observation. The core component of the architecture, namely the observer, is itself designed as a Web Service to make it available to all interested entities. Moreover, by reusing the Web Services paradigm, we minimize the required knowledge and skills to interact with it. I presented the two possible architectures: mono-observer architecture and the multi-observer architecture. The first one is used if the WSUO is a non-composite Web Service, while the multi-observer architecture is suitable for composite Web Services.

In the next chapter, I will tackle the first requirement of the architecture namely how to specify the expected behavior of Web Services. The second requirement, traces' collection, will be thoroughly studied in Chapter 6.

Chapter 5
Web Services' Behavior Description

I hear and I forget; I see and I remember;
I do and I understand -
Confucius

In order to issue a verdict (correct/faulty) regarding the correctness of an observed trace, a passive observer must have unambiguous description of the expected behavior of the WSUO. This behaviors' information can be of concern to three facets: 1) structure of requests and responses, 2) content and sequence of requests and responses, and 3) QoWS attributes of the WSUO.

In Web Services' interactions, the structure of requests and responses is highly typed and precisely defined in WSDL documents. For the content and sequence of requests, this can be represented by a knowledge base (expert systems-based observers) or a formal model of the WSUO (model-based observers).

Formal models have many advantages over knowledge bases [33]. First of all, expert systems rely on human expertise and are more appropriate for systems that have been encountered previously. Second, formal models can be useful during different phases of the Web Service development life cycle: specification, design, validation, automatic generation of source code and executable test cases... In fact, during the life cycle of a Web Service [90], the initial requirements go through certain steps conducting to a (hopefully) well designed

---

* Results from this chapter have been published in [85], [86], [87], [88], [89], [90], and [91].

and developed Web Service. During some of these steps, the use of formal methods can allow automation of many activities in the development lifecycle and some methods provide even proofing engines.

Once constructed, a formal model of WSUO can be used in many tasks: automatic source code generation, automatic test cases generation, and passive observation as illustrated by Figure 5.1. For these reasons, model-based observers will be considered in remaining parts of this thesis.

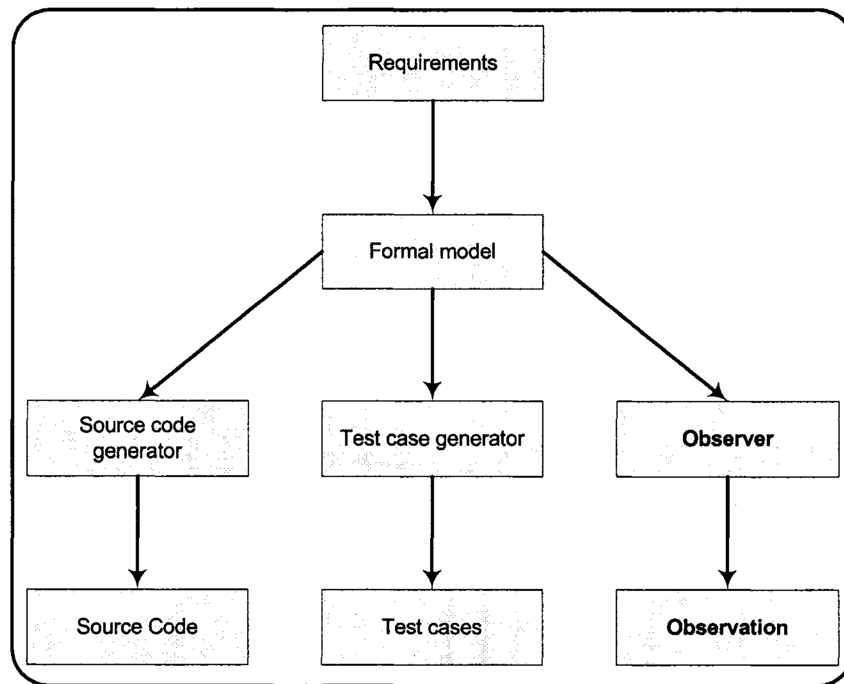

Figure 5.1 Extending the use of formal models by/for observation

Formal methods offer different levels of abstraction which help in coping with the complexity of systems' requirements and descriptions. Different levels of abstraction of formal methods imply different complexities: the more a model is expressive, the higher is its com-

plexity, and the less this model is studied in the literature and related work. As will be discussed in following chapters, existing approaches have to be improved.

Description of QoWS attributes can be provided to a passive observer in many different ways. This information can be described in a separate document, embedded within the description of functional behavior, or as an extension to WSDL document.

In this chapter, I will present models for describing Web Services' behaviors that will be used in following chapters. The next section presents information that is already provided by the programming stack of SOA, namely WSDL documents, which are available for all Web Services. Section 5.2 discusses specification models for functional aspects while section 5.3 discusses specification of QoWS properties.

5.1 WSDL

The SOA requires that each Web Service be described by a WSDL document. This document contains the list of available operations, their signatures, supported protocol, and end points. Roughly speaking, it holds all the information that a client needs to bind to the associated Web Service. A WSDL document basically defines the following elements:

- **Types**: this is a container to define additional complex data types using some type system (such as XML Schema Datatypes (XSD)).
- **Message**: it consists of an abstract-typed definition of the data being communicated. It specifies which XML data types constitute various parts of a message, both input and output parameters of operations.
- **Operation**: it states, for each operation, the input and output data and the order in which they should be used in an invocation.

- **Port Type:** this element defines the operations actually supported by the Web Service and specifies the XML messages that can appear on input and output data flows. It can be seen as method signature/prototype in programming languages.
- **Binding:** it describes the protocols, data format, and other issues for binding to the Web Service.
- **Port:** this element, also known as endpoint, is defined as a combination of a binding and a network address.
- **Service:** this is a collection of related Ports/endpoints.

Figure 5.2 shows how these elements fit in the structure of a Web Service description. This figure is a partial view of the WSDL description of the Call Control Web Service that will be used in the case study of section 6.2.

The information brought by the WSDL document of the WSUO allows the detections of two types of faults:

1. Input Type Fault (ITF): an input type fault is observed when a method is invoked with a wrong number and/or wrong types of parameters with regards to its signature published in the WSDL.
2. Output Type Fault (OTF): this fault occurs if the type of the returned result is different from the type expected in the WSDL document.

```
<?xml version="1.0" encoding="UTF-8"?>

 <wsdl:message name="initiateConfRequest">
   <wsdl:part name="addresses" type="intf:ArrayOf_xsd_string"/>
   <wsdl:part name="mediatype" type="xsd:string"/>
   <wsdl:part name="conftype" type="xsd:string"/>
   <wsdl:part name="duration" type="xsd:int"/>
   <wsdl:part name="expectedUsers" type="xsd:int"/>
   <wsdl:part name="callID" type="intf:ArrayOf_xsd_string"/>
 </wsdl:message>

 <wsdl:portType name="ConferenceService">
   <wsdl:operation name="addUser" parameterOrder="userAddresse callID">
     <wsdl:input message="intf:addUserRequest" name="addUserRequest"/>
     <wsdl:output message="intf:addUserResponse" name="addUserResponse"/>
   </wsdl:operation>
  </wsdl:portType>

 <wsdl:binding name="ConferenceServiceSoapBinding" type="intf:ConferenceService">
   <wsdlsoap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
   <wsdl:operation name="addUser">
     <wsdlsoap:operation soapAction=""/>
     <wsdl:input name="addUserRequest">
       <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" name-
space="http://142.133.72.116:7777/axis/services/ConferenceService" use="encoded"/>
     </wsdl:input>
     <wsdl:output name="addUserResponse">
       <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" name-
space="http://142.133.72.116:7777/axis/services/ConferenceService" use="encoded"/>
     </wsdl:output>
   </wsdl:operation>
 </wsdl:binding>

 <wsdl:service name="ConferenceServiceService">
   <wsdl:port binding="intf:ConferenceServiceSoapBinding" name="ConferenceService">
     <wsdlsoap:address location=
"http://142.133.72.116:7777/axis/services/ConferenceService"/>
   </wsdl:port>
 </wsdl:service>
</wsdl:definitions>
```

Figure 5.2 Partial WSDL of the Call Control Web Service

These types of faults are likely to happen when the WSDL document is not up-to-date. This might be the case in two situations:

1. The provider modifies the data type signature of published operations without publishing the new WSDL.
2. The client does not look periodically for updated WSDL documents in the UDDI registry and uses outdated documents.

Even though some programming languages (e.g. C++) have the ability to cast between specific types (e.g. int to/from integer), the SOAP infrastructure, at the client's side or the Web Service's side, does not have the mentioned capability. Proper conversion has to be accurately performed by Web Services and their calling clients.

The next section will present three models that can be used in a model-based observer to detect functional faults.

5.2 Description of the functional aspects

In model-based observers, the types of functional faults that can be detected depend on the used model. Among all formal models that have been used for testing, in this thesis, we are going to focus on two of them with different levels of abstractions: FSM and EFSM. These models need first to be described in the same description languages used in the SOA paradigm. The following two subsections propose how these two models can be represented in XML documents that a Web Service observer can process to detect functional misbehaviors. Subsection 5.2.3 discusses the functional behavior's specifications for composite Web Services using BPEL.

5.2.1 FSM

The FSM model has been used for many decades for modeling the behavior of communication protocols [24]. As mentioned earlier in Section 2.5.2, it is an appropriate concept to formally represent control part of systems which leads to the detection of classes of faults. In our case, we will address the following classes of faults: input faults (IF) and output faults (OF).

An FSM can be represented as an XML document as illustrated in Figure 5.3. The root of the document *(fsm)* has an attribute *(name)* and a set of children which represents states. The name is a textual description of the machine or its associated Web Service. Each child has a name, the attribute *(initial)*, and a set of *transitions*. The name is a textual description of the state while the attribute (initial), if set to YES, specifies that this is the initial state of the machine. A transition has four attributes: *ID*, *input*, *output*, and *next*. The first attribute is a textual description of the transition; the second attribute identifies the event that triggers this transition if the machine is in the associated state; the third attribute is the output generated by firing that transition; and the last attribute specifies the state that the machine will reach after firing the transition.

5.2.2 EFSM

EFSM is a richer model than FSM since it allows the expression of data such as variables and parameters. In an EFSM model, input and output events are parameterized and carry data that transitions manipulate in addition to local variables. Since an EFSM model is an extended version of FSM models, the XML representation of an EFSM model extends the

XML representation of an FSM model (Figure 5.3) with two attributes: *predicate* and *assignments*. The first attribute indicates a Boolean expression that should evaluate to TRUE in order to fire this transition. The second attribute represents the set of data manipulation to be performed while firing the transition.

```
<fsm name="Name of FSM/Web Service">
        <state name="State1" initial="YES">
                <transition ID="t1"     input="Input1" output="Output1" next="State2"/>
                <transition ID="t2"     input="Input2" output="Output2" next="State3"/>
        </state>
        <state name="State2" initial="NO">
                <transition ID="t3"     input="Input3" output="Output3" next="State3"/>
        </state>
        <state name="State3" initial="NO">
                <transition ID="t4"     input="Input4"output="Output4" next="State2"/>
        </state>
</fsm>
```

Figure 5.3 XML representation of an FSM machine

FSM and EFSM machines are generally manually generated from initial specifications. However, there have been many works to help in automatic generation from certain specification languages. While this generation is out of the scope of this thesis, the reader can refer to some of these methods in ([92], [93], [94], [95], [96], and [97]).

```
<efsm name="Name of EFSM/Web Service">
        <state name="State1" initial="YES">
                <transition     ID="t1"
                                input="Input1"
                                predicate="true"
                                assignments="x:=0;y:=0;z:=0"
                                output="Output1"
                                next="State2"/>
                </transition>
                <transition     ID="t2"
                                input="Input2"
                                predicate="X<3"
                                assignments="x:=2;y:=7"
                                output="Output2"
                                next="State3"/>
                </transition>
        </state>
</efsm>
```

Figure 5.4 XML representation of an EFSM machine

5.2.3 BPEL

When the WSUO is a composite Web Service, its BPEL description provides complete information on how the WSUO is supposed to behave. BPEL, based on XML, can be used to specify business processes and business interaction protocols. A BPEL process defines clearly its interactions with its partners: its client and its basic Web Services. In general, a BPEL process defines the following tags as depicted in Figure 5.5.

```
<process ...>
        <partners>
                ...
        </partners>
        <containers>
                ...
        </containers>
        <correlationSets>
                ...
        </correlationSets>
        <faultHandlers>
                ...
        </faultHandlers>
        <compensationHandlers>
                ...
        </compensationHandlers>
        <!-- Activities -->
</process>
```

Figure 5.5 Structure of BPEL process

- **Partner**: this element defines partners of the BPEL process. These include the client and basic Web Services this process interacts with.
- **containers**: this defines the data used by the process for internal computations, to pass information to partners, etc
- **correlationSets**: this tag is used to support asynchronous interactions so that requests and responses can be correlated.

- **Fault Handlers**: this defines a fault tolerant capability of BPEL in which a process can specify the code to execute when, for example, undoing an action that failed or whenever asked to rollback.

Activities: this is the main element in a BPEL process. It specifies what the process actually does, both internal behavior and interactions with partners. An activity can be one of the following actions: invoke, receive, reply, assign, wait, throw, compensate, terminate, flow, switch, while, sequence, and pick.

Figure 5.6 shows a BPEL process and four partners and Figure 5.7 shows a scenario where this process receives a request from partnerLink client. It has to invoke partnerLink "Basic1" to satisfy this request, assign appropriate variables, and then return a response to the client.

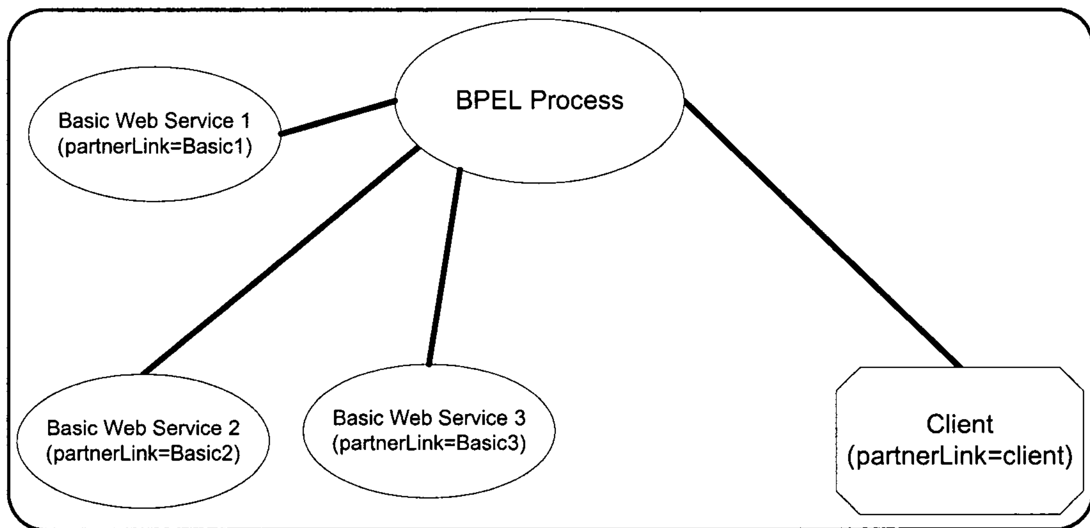

Figure 5.6 BPEL process with partnerLinks

```
<sequence name="main">
        <!-- Receive input from requester -->
                <receive name="receiveInput" partnerLink="client"
                        portType="tns:Composed_BPEL" operation="process"
                        variable="input" createInstance="yes"/>
        <!-- Assign appropriate variables -->
                <assign name="assign-1">
                        <copy>
                                <from expresion="concat(bpws:getVariableData(
                                        'input','payload',
                                        '/tns:Composed_BPELRequest/tns:input'), ' ')">
                                </from>
                                <to variable="output" part="payload"
                                query="/tns:Composed_BPELResponse/tns:result"/>
                        </copy>
                </assign>
        <!-- Invoke basic Web Service -->
                <invoke name="invoke-1" operation="process_0"
                        inputVariable="in"
                        outputVariable="out"
                        partnerLink="Basic1"
                        portType="ns0:Basic_Soap"/>
        <!-- Return result to requester -->
                <reply name="replyOutput" partnerLink="client"
                        portType="tns:Composed_BPEL"
                        operation="process"
                        variable="output"/>
</sequence>
```

Figure 5.7 Example of BPEL activities tag

A BPEL process defines behavior of the WSUO with the client and its behavior with its basic Web Services. Operations receive and reply defines invocation of the composite Web Service by the client and the operation invoke defines its interactions with its basic Web Services. FSM and EFSM models can be derived from a BPEL process for both sides of interactions (basic Web Services and client).

Once the functional aspect of Web Services' behaviors have been specified as FSM or EFSM machines, the non-functional aspects (QoWS) must be specified too. The next section discusses how this information can be defined and represented in such a way it can be handed to an observer.

5.3 QoWS aspects

QoWS consists of a set of factors or attributes such as processing time, response time, reliability, availability, accessibility, etc. In SOA, a lot of work is taking place to allow both Web Services' providers and their clients to define and concisely use QoWS description to enable publication, discovery, and usage of QoWS attributes.

5.3.1 QoWS attributes

The first step in extending SOA with QoWS is the definition of its attributes. In this thesis, we will focus on the following attributes:

- Processing Time (PT): this is a measure of the time a Web Service takes between the time it gets a request and the moment it sends back the corresponding response. PT is computed at the Web Service's provider side.

- Maximum Processing Time (MxPT): this is the maximum time the Web Service should take to respond to a request.
- Minimum Processing Time (MnPT): this is the minimum time the Web Service should take before responding to a request. Unlike PT which is a dynamically computed attribute, MnPT and MxPT are statically defined and: MnPT $\leq$ PT $\leq$ MxPT.
- Response Time (RT): it consists of the time needed between issuing a request and getting its response. It is measured at the client's side to include the propagation time of requests and responses.
- Maximum Response Time (MxRT): this is the maximum accepted time, for the client, between issuing a request and getting its response.
- Minimum Response Time (MnRT): this is the minimum time, for the client, between issuing a request and getting its response. This attribute in unlikely to be used since the client is usually more interested in MxRT. For the client: RT $\leq$ MxRT must always be satisfied.
- Availability: this is a probability measure that indicates how much the Web Service is available for use by clients. It can also consist of the percentage of time that the Web Service is operating.
- Service Charge (SC): for accounting management purposes, this attribute defines the cost a client will be charged for the Web Service's utilization. SC can be estimated by operation, type of requests, period of utilization, session, or by volume of data.

- Reputation: this is a measure of Web Services' credibility. It depends basically on previous end users' experiences while using the Web Service. Different users may have different opinions on the same Web Service. The reputation value can be given by the average ranking given to the service by several users.

MnPT, MxPT, availability, and SC are related to profiles of users of the Web Service. This profiling is based on the type of subscriptions of users and/or the QoWS they are paying for. For example, a gold-subscribed (MnRT = 0) user must be served quicker than a bronze-subscribed user (MnRT > 1ms).

Specification of the QoWS attributes discussed above allows the detection of the following misbehaviors:

- MxPT violation: if the Web Service does not respond before this duration expires, it's considered as a violation.
- MnPT violation: if the Web Service responds before MnPT duration expires, it's considered as a violation.
- MxRT violation: if the client is still waiting for a response after MxRT time unit, this is considered as a violation.
- MnRT violation: if the response arrives before MnRT expires, it's considered as a violation.
- RT violation: whenever RT becomes higher than MxRT, a QoWS violation is detected.
- SC violation: the client should pay for the used Web Service and the associated profile; otherwise, the manager should be notified about this violation.

In order to detect QoWS violations, an observer must have detailed information on different attributes defined above. This information can be handed to the observer in different ways: in a dedicated document, embedded with functional behavior specification, or as part of the WSDL document. These possibilities are discussed in the following sub-sections.

5.3.2 QoWS attributes in a dedicated document

When the manager of the Web Service is only interested in QoWS management, it is appropriate to hand this information in a dedicated document. The size of this document will be as minimal as possible and the time required by the observer to process it will be reduced.

Figure 5.8 shows and example of specification of some QoWS attributes in a dedicated XML-based document. For each operation and profile, QoWS attributes are defined.

```
<QoWS name= "Name of Web Service">
        <Profile name="GOLD">
                <operation      name= "op1"
                                MnPT = NULL
                                MxPT = 10ms
                                SC= "$10"
                </operation>
                <operation      name= "op2"
                                MnPT = NULL
                                MxPT = 50ms
                                SC= "$60"
                </operation>
        </Profile>

        <Profile name="SILVER">
                <operation      name= "op1"
                                MnPT = 10ms
                                MxPT = 30ms
                                SC= "$5"
                </operation>
                <operation      name= "op2"
                                MnPT = 20ms
                                MxPT = 80ms
                                SC= "$40"
                </operation>
        </Profile>
</QoWS>
```

Figure 5.8 QoWS specification in dedicated document

5.3.3 QoWS specification embedded with functional behavior

Incorporating the specification of QoWS attributes into the specification of the functional behavior of the Web Service is practically efficient when the manager is targeting both functional and QoWS monitoring. In fact, the observer will get one document from which it builds a complete view of the expected behavior of the Web Service, regarding both functional and QoWS aspects.

Figure 5.9 shows how an FSM model can be extended to reflect QoWS attributes. We will designate the new model by FSM+. For each transition, different QoWS are specified for different profiles. The same approach can be used to incorporate QoWS attributes into EFSM models.

5.3.4 QoWS attributes in WSDL

Discovery and selection of Web Services might be based on QoWS offered by providers and required by clients. Many providers are likely to offer Web Services providing similar functionalities; however, with quite different QoWS attributes (how much time is it available? How cheap/expensive it is? What are its PT, MnPT, and MxPT?).

To allow QoWS-aware discovery and selection of Web Services, QoWS attributes should be available within the WSDL document. The client indicates preferences in terms of Quality of Service when issuing a FIND inquiry to the registry. The registry returns as a result a list of available Web Services providing functional operations with required QoWS.

```
<fsm+ name="Name of FSM/Web Service">
        <state name="State1" initial="YES">
                <transition     input="Input1"
                                output="Output1"
                                <Profile name="GOLD">
                                        MnPT = NULL
                                        MxPT = 10ms
                                        SC= "$10"
                                </Profile>
                                <Profile name="SILVER">
                                        MnPT = 10ms
                                        MxPT = 30ms
                                        SC= "$5"
                                </Profile>
                                ...
                </transition>
                ...
        </state>
        ...
</fsm+>
```

Figure 5.9 FSM+ specification

WSDL documents are required for observation of their associated Web Services (section 5.1). Additionally, the observer can get QoWS attributes' values from the WSDL document whenever QoWS-aware Web Services are being observed.

Figure 5.10 illustrates embedded QoWS attributes in the definition of the operation tag within the WSDL document initially presented in section 5.1.

```
<?xml version="1.0" encoding="UTF-8"?>
 <wsdl:portType name="ConferenceService">
  <wsdl:operation name="addUser" parameterOrder="userAddresse callID">
   <wsdl:input message="intf:addUserRequest" name="addUserRequest"/>
   <wsdl:output message="intf:addUserResponse" name="addUserResponse"/>
                    <Profile name="GOLD">
                         MnPT = NULL
                         MxPT = 10ms
                         SC= "$10"
                    </Profile>
                    <Profile name="SILVER">
                         MnPT = 10ms
                         MxPT = 30ms
                         SC= "$5"
                    </Profile>
  </wsdl:operation>
 </wsdl:portType>
```

Figure 5.10 QoWS in WSDL document

5.4 Summary

Misbehavior detections are based on information available to the observer. This information must define, unambiguously, the expected behavior of the observed Web Service. Expected behavior covers both functional and QoWS aspects.

In this chapter, we proposed a set of possible descriptions that can be used to offer the needed information on Web Services' expected behaviors to observers. First, we demonstrated that we can obtain information from WSDL on available operations within a Web

Service and their precise data type signatures. Second, we presented XML representations of FSM and EFSM models for specification of functional behavior. Finally, we showed how QoWS attributes can be specified as an extension FSM/EFSM machines, as a dedicated document, or embedded to WSDL.

In the next chapter, we will study different mechanisms for traces' collection. We will also discuss the overhead introduced by different components of the management architecture.

Chapter 6
Management Architecture for Simple Web Services

The difference between practice and theory
is a lot bigger in practice than in theory -
Peter van der Linden

The main objective of passive observation is misbehavior's detection. To do so, we need to collect traces at appropriate points of observation and analyze them with respect to the specified behavior. This chapter is dedicated to the study and comparison of mechanisms that allow efficient collection of traces. The comparison is based on some criteria such as network load, availability of resources, and required participation from different actors. In this chapter, we will also evaluate the effectiveness of the management architecture in terms of misbehavior's detection through a real case study of a non-composite Web Service.

6.1 Communication between components

One of the design keys to investigate in a passive observation context is how to provide the observer with all exchanged request/response sequences. The mechanisms that can be used depend essentially on who is interested in the observation: the client, the provider of the Web Service, or a third party; and also on the location of the client, the Web Service, points of observation, and/or the observer. What might be the overhead of the proposed mechanism and the feasibility of such mechanism in that particular context?

* Results from this chapter have been published in [85] and [86].

Potential solutions range from multicast addresses to sniffer-like dispatcher, and can be divided into two main groups: *fixed code* and *mobile code*. Fixed code mechanisms include instrumentation of the client and/or the Web Service, multicast, SNMP agents/managers, and dispatchers. Mobile agents are an example of mobile code approaches that can be used for traces' collection. These mechanisms differ in their specific requirements (resources, configuration...) and generated overhead (CPU and network load).

6.1.1 Client/Web Service instrumentation

The easiest way to forward exchanged traces to the observer is through participation of the Web Service, its client, or both of them. For example, if the observation is requested by the client and the source code of the client application is available, few lines of code can be added to duplicate all sent requests and received responses and send them to the observer (Figure 6.1). This approach, however, supposes that the source code of the Web Service or the client application is available and the provider or the client is willing to instrument it in order to participate in the observation. Such types of Web Services are designed with test and observation capability that can be enabled/disabled when needed. This technique is known as Design for Testability.

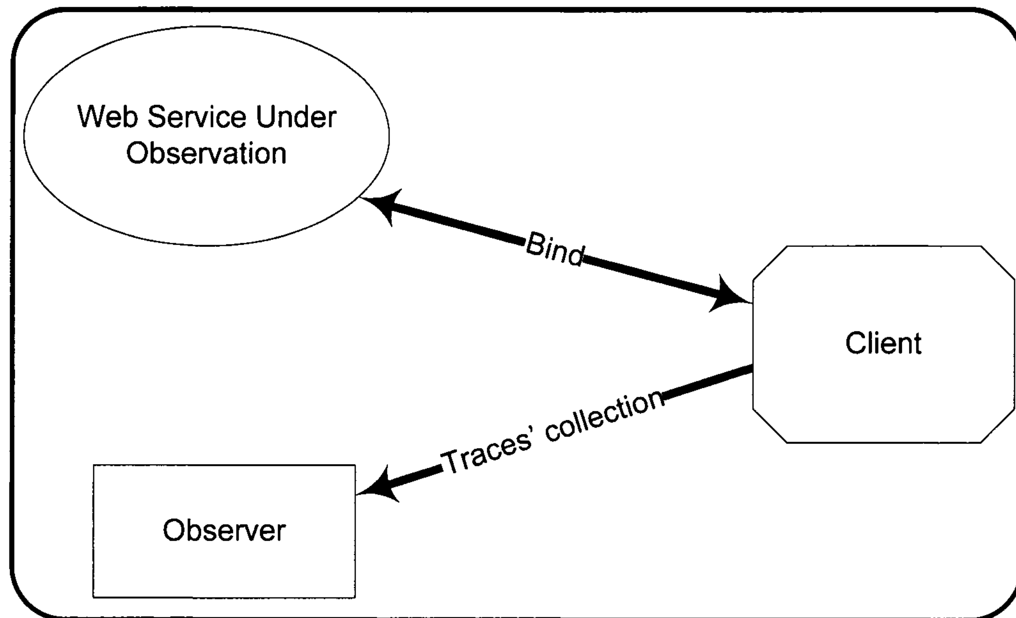

Figure 6.1 Client's instrumentation for traces' collection

6.1.2 Multicast

Multicast communication is another possibility wherever multicast routing features are supported. In this type of communication, hosts can join or leave a multicast address and those who joined will receive the traffic related to that multicast address.

When the communication between a Web Service and a client is monitored, they can send their packets to a multicast addresses rather than unicast addresses. The observer, Web Service, and its client can subscribe to the same multicast address. Using this multicast address, the observer will get a copy of all exchanged request/response pairs (Figure 6.2). The observer leaves the multicast addresses once the observation period ends. This solution supposes that the network infrastructure supports multicast traffic.

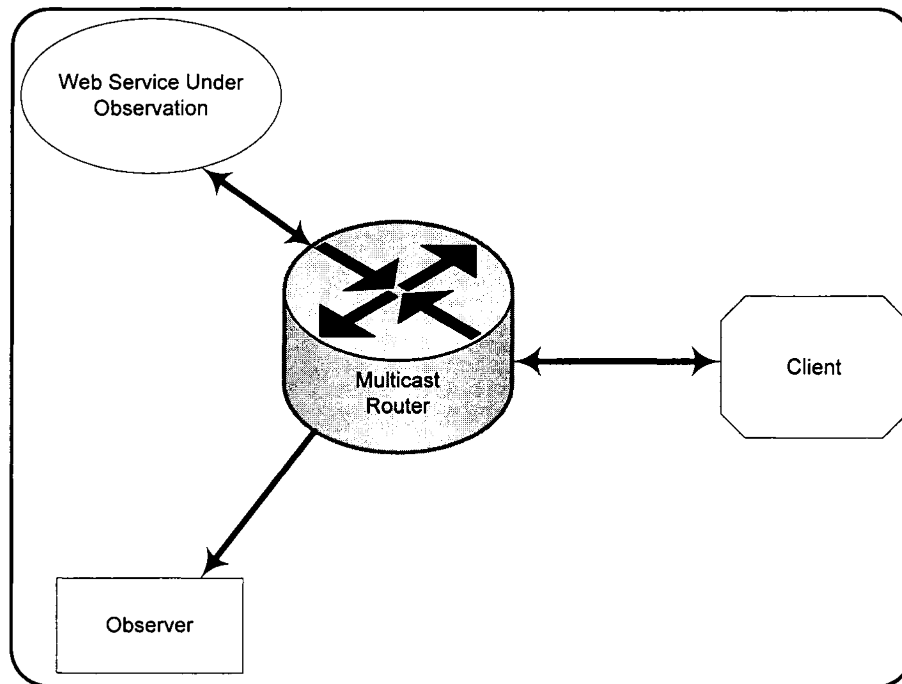

Figure 6.2 Multicast for traces' collection

6.1.3 Dispatcher

This solution is quite similar to the multicast approach and makes use of sniffers-dispatchers (spy or man-in-the-middle) (Figure 6.3). Such dispatchers sniff the traffic on the network, duplicate and then forward to the observer all messages to/from a specific host (Web Service and/or client).

Dispatchers can operate at two different levels: IP level and TCP level. Even if a standalone application can be developed for IP-based dispatcher, most of actual routers and firewalls can be easily configured to perform this task. Traces retrieved by an IP-based dispatcher will be quite big since they are sniffed at a low level. Moreover, the observer must process these traces (IP packets) to figure out invocations' information (SOAP messages).

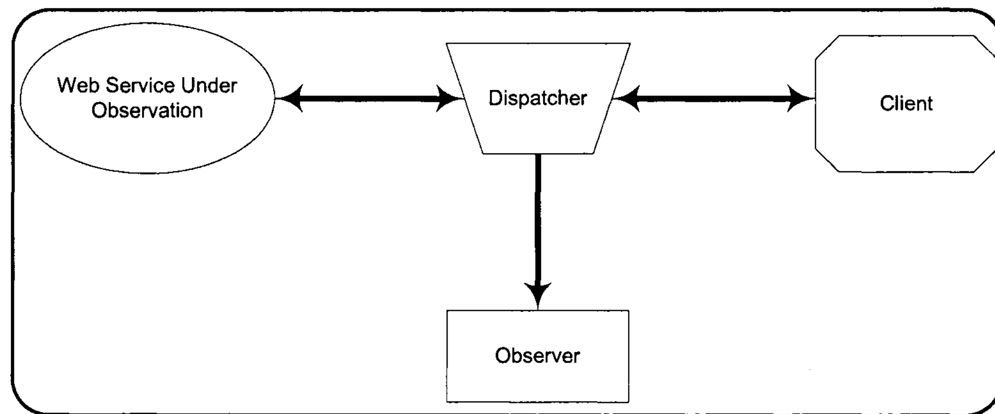

Figure 6.3 Sniffers-Dispatchers for traces' collection

To use a TCP-based dispatcher, the client should address requests to the dispatcher and not to the Web Service. The dispatcher forwards the request to the Web Service and a copy to the observer. When responses are received, they are sent to the client and copied also to the observer. Since TCP is a connection oriented transport protocol, a TCP-based dispatcher cannot be used for communication already in progress between the Web Service and the client. It can only be used for new invocations of Web Services.

It should be noted that operations of both IP-based and TCP-based dispatcher are transparent to the Web Service and the client. In fact, SOAP messages are carried by HTTP which is based on TCP. SOAP infrastructure does not deal with establishment/termination of connections. However, TCP-dispatcher introduces a delay that is usually very small compared to processing time and response time.

6.1.4 SNMP

SNMP can be used as a way to collect traces. This mechanism makes use of the Management Information Base (MIB) located at the client and/or the Web Service side. Whenever a new

event is added to the MIB[2], an SNMP agent sends it in an unsolicited trap to the observer. The observer acts exactly as an SNMP manager in this configuration. Figure 6.4 shows an MIB and SNMP agent located at the client's side.

The SNMP mechanism requires a complete MIB and an agent. Usually, the MIB contains information on actual status of the managed application and, in the best case, a subset of exchanged information. For SNMP-based traces' collection, the MIB must hold all invocations with their parameters.

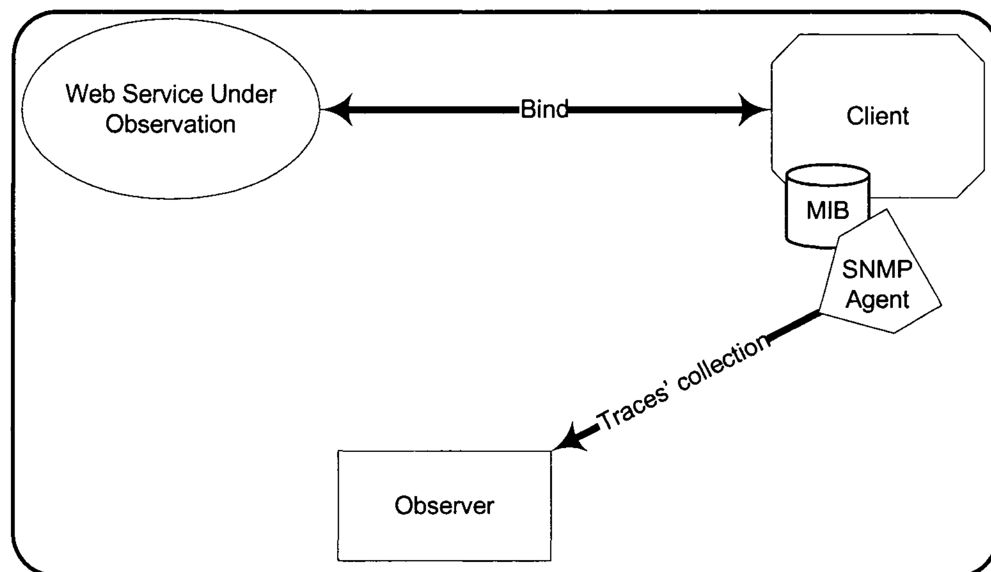

Figure 6.4 SNMP agents for traces' collection

[2] There is a language misuse when we say that the agent or the application writes/reads to/from the MIB. The MIB describes, in ASN.1 notation, information on managed objects. Effective values and log entries are however written/read to/from a dedicated file/database.

6.1.5 Mobile code

In the mechanisms presented above (sections 6.1.1 to 6.1.4), management capabilities are all put in the Web Service observer provider's side, and the problem to solve was to collect and move traces to the observer. If the WSO is located far away from the point where the traces are being collected, the effectiveness of these mechanisms might be affected by the performance of the overall network. Congestions and network links' failures can stall management procedures of the management architecture.

A completely different approach is to move the observer itself to the Web Service/client side instead of forwarding traces. This is made possible by using mobile agent technologies [98]. The WSO, when invoked, can generate and send a mobile agent to the calling mobile platform. For example, if the Web Service's provider invokes the WSO, a mobile agent will be sent back. This agent observer will reside on the same local network, but not necessarily on the same node as the Web Service. The agent observer can then get traces *locally* using one of the previously presented mechanisms (Subsections 6.1.1 through 6.1.4).

This approach requires a mobile agent platform to be available but the Web Service itself does not have to be running on a mobile platform. The mobile platform can reside on another node within the network as long as it can access exchanged traces. The network load in this case is low and is limited to the mobility of the mobile agent (independent from the size of traces).

6.1.6 Discussion

Deciding on what traces' collection mechanism, from previously presented, to use depends on many factors. First of all, the availability of a specific mechanism is necessary for its utilization.

Dispatchers can be provided by the provider of the Web Service observer. The manager will then deploy and configure it in an appropriate location in the network. Client or Web Service instrumentation cannot be used unless the source code is available and can be modified, re-compiled, and re-deployed or designed from the beginning with the desired capability. The multicast approach is possible if the network infrastructure supports multicast routing and the observer, as well as the client, the Web Service can join and leave multicast addresses. The SNMP mechanism requires a complete MIB and an SNMP agent. While SNMP agents are more or less easy to develop and deploy, a complete MIB must be designed and deployed. A complete MIB holds all and every interactions and its parameters between the Web Service and its client. For example, if the invocation returns a huge file, the MIB should keep trace of this file. Mobile code requires a mobile platform which can be downloaded and configured offline.

In the case where all mechanisms are available, their comparison can be based on five criteria: CPU and memory utilization, level of participation of client and/or Web Service's provider, required changes to client and/or Web Service source code, generated network load, and additional delay. Section 6.2.4 gives an experimental evaluation of each of these criteria for all mechanisms.

The next section presents an implemented case study where our management architecture has been used. It will also discuss the overhead introduced by management operations and the effectiveness of the management architecture in detecting misbehaviors. In this first prototype, we focus only on functional behavior; QoWS issues will be considered in the case study presented in the next chapter (Chapter 7).

6.2 Case study

To illustrate the applicability of our management approach, a Call Control Web Service [99] exposing some of the functionalities of Parlay/Call Control [100], has been observed using FSM models. This Web Service provides mainly the conferencing parts of the parlay specification. For the sake of this experimentation, all traces' collection mechanisms are implemented and evaluated except multicast and IP-based dispatchers. Multicast routing is not widely supported in networks; which reduces the possibility of its utilization for traces' collection. As explained above, IP-dispatchers are more complex and generate bigger traces since they operate at a low level.

Table 6.1 presents the functionalities that the Call Control Web Service, deployed on Ericsson Application Server, exposes at a high level of abstraction [99]. The first column shows a description of the Web Service functionalities and the second column shows the corresponding published interface:

Table 6.1. Exposed functionalities and their corresponding published interfaces

Functionality	Published interface
Initiate a dial-out conference with n users	initiateConf
Add users to an ongoing conference	addUser
Move users from a conference to its sub-conferences or from a sub-conference to another sub-conference	moveUser
Remove users from an ongoing conference	removeUser
Initiate sub-conferences	initiateSubConf
End sub-conferences	endSubConf
End conferences	endConf

The XML document describing the FSM machine is illustrated in Figure 6.5 and its graphical representation in Figure 6.6.

```xml
<fsm name="Call Control">
    <state name="Config" initial="YES">
        <transition input="config_Invalid"    output="FALSE" next="Config"/>
        <transition input="config_Valid"      output="TRUE" next="Idle"/>
    </state>
    <state name="Idle" initial="NO">
        <transition input="initiateConf_Invalid"    output="FALSE" next="Idle"/>
        <transition input="initiateConf_Valid"      output="TRUE" next="Ready"/>
    </state>
    <state name="Ready" initial="NO">
        <transition input="endConf_Valid"                 output="TRUE"  next="Idle"/>
        <transition input="endConf_Invalid"               output="FALSE" next="Ready"/>
        <transition input="addUser_Valid"                 output="TRUE"  next="Ready"/>
        <transition input="addUser_Invalid"               output="FALSE" next="Ready"/>
        <transition input="removeUser_Valid"              output="TRUE"  next="Ready"/>
        <transition input="removeUser_Invalid"            output="FALSE" next="Ready"/>
        <transition input="moveUser_Valid"                output="TRUE"  next="Ready"/>
        <transition input="moveUser_Invalid"              output="FALSE" next="Ready"/>
        <transition input="initiateSubConf_Valid"         output="TRUE"  next="Ready"/>
        <transition input="initiateSubConf_Invalid"       output="FALSE" next="Ready"/>
        <transition input="endSubConf_Valid"              output="TRUE"  next="Ready"/>
        <transition input="endSubConf_Invalid"            output="FALSE" next="Ready"/>
    </state>
</fsm>
```

Figure 6.5 XML description of the Call Control FSM machine

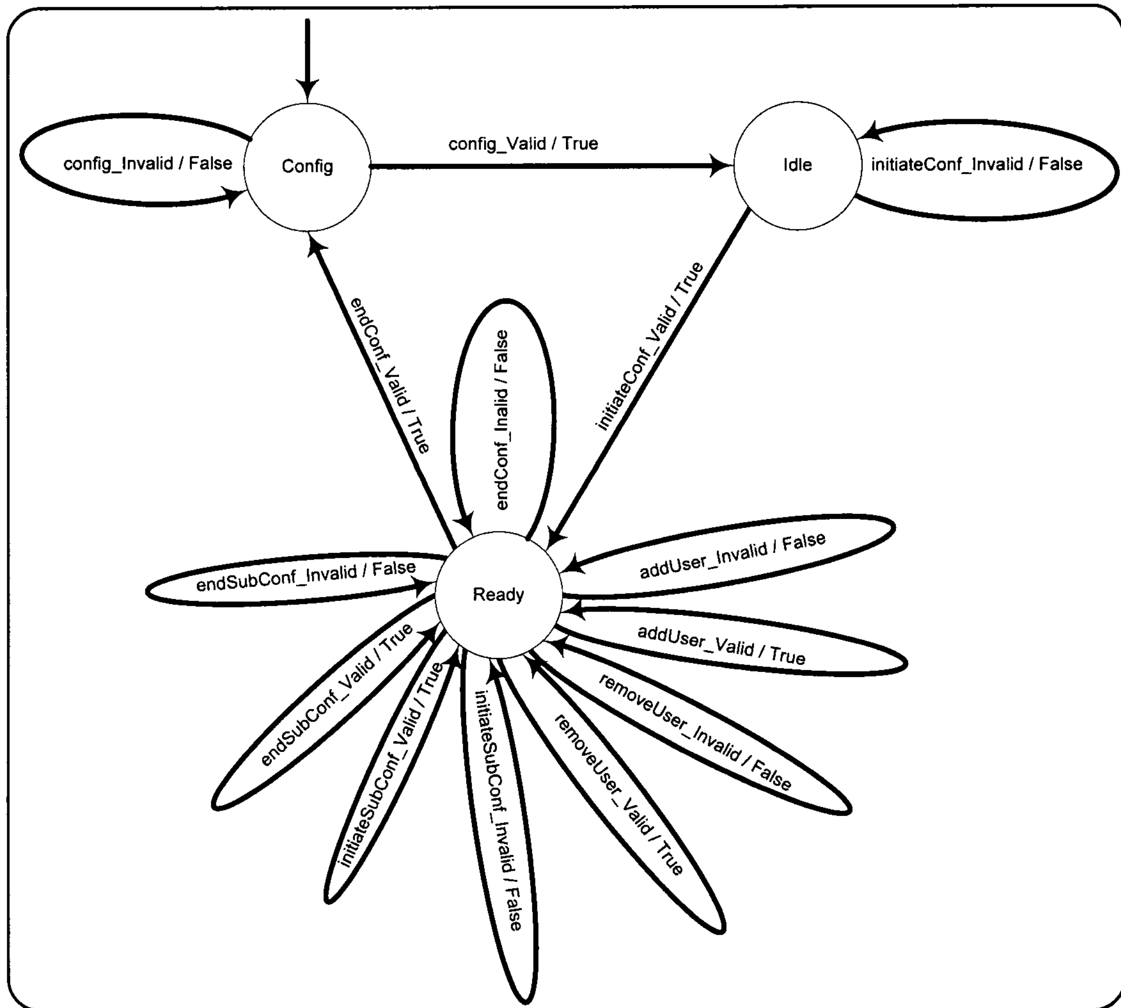

Figure 6.6 Graphical representation of the Call Control FSM machine

6.2.1 Web Service Observer

The Web Service Observer, developed on BEA WebLogic Server [2], offers 4 operations to the manager: "initObserver", "sendRequestToObserver", "sendResponseToObserver", and "sendRequestResponseToObserver". The invocation is performed using the "initObserver" interface by providing the observer with the FSM of the Call Control Web Service, time to

start observation, and time to stop observation. The observer then parses the XML document and uses a special data structure to store the FSM. If the parsing succeeds, the observer is then ready to process requests and responses messages. In our architecture, requests stands for messages from the client to the Web Service and response for messages sent back by the Web Service to the client. The trace's collection entity has two options:

1. Send observed requests and responses separately to the observer using "sendRequestToObserver" and "sendResponseToObserver" interfaces, or
2. Combine a request and its response and send them using the interface "sendInputOutputToObserver".

The observer checks the validity of each trace and returns a decision to the manager whenever a functional fault is detected.

6.2.2 Client

A client application offers, through its graphical interface (Figure 6.7), the possibility to invoke any operation from those offered by the Call Control Web Service. For each operation, the client can decide between a valid and invalid invocation. This selection is imposed by the FSM-based observers, which do not consider data and are then unable to process parameters of invoked operation and decide between valid and invalid parameters. As indicated by the FSM machine in Figure 6.6, the Web Service should return the output "true" if the operation is valid or "false" if the operation is invalid, otherwise a fault is detected.

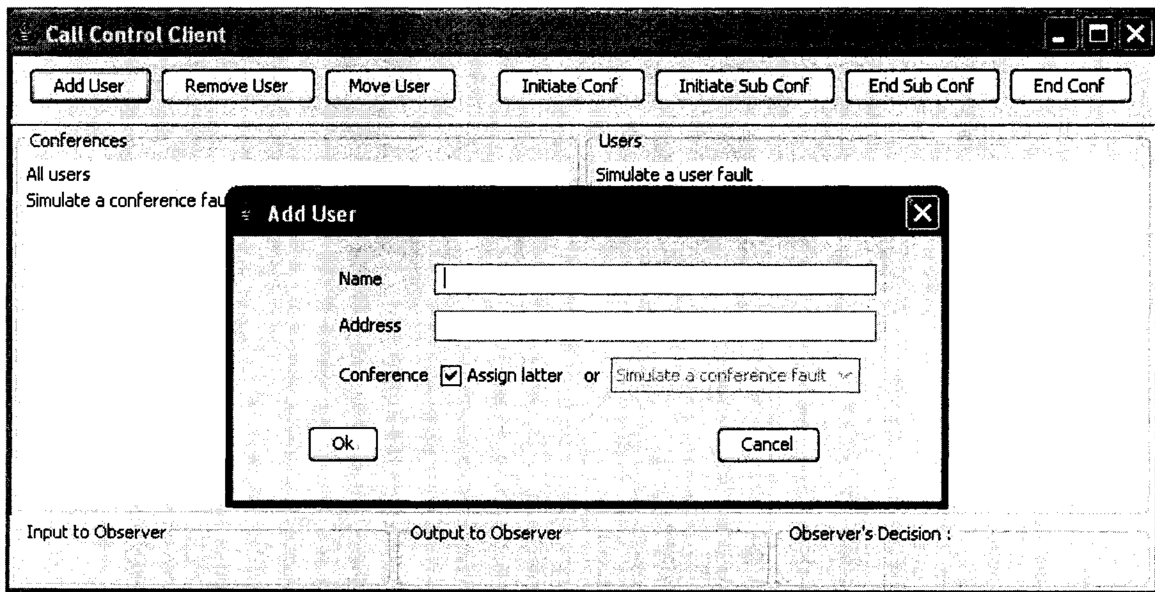

Figure 6.7 Client application GUI

The client application is instrumented and can be used as a mean of traces' collection. In this case, the three text zones at the bottom of the GUI show requests, responses, and the decision of the observer.

6.2.3 Detection capabilities

The following table reports some of the scenarios that have been executed with their observed verdicts.

Table 6.2 Executed scenarios and their corresponding verdicts

Targeted operation	Scenario (Valid/Invalid)	Verdict	Comments
initiateConf	Valid (With initial users)	Pass	
initiateConf	Valid (List of initial users empty)	Fail	No response and no decision received from the observer
initiateConf	Invalid (wrong conference type dial out/dial in)	Fail	A dial out conference is initiated, a fault is detected
initiateSubConf	Valid	Pass	
initiateSubConf	Invalid (initiate a sub-conference before initiating a conference)	Pass	The observer detects an INPUT fault
addUser	Valid	Pass	
addUser	Invalid (add a user before initiating a conference)	Pass	The observer detects an INPUT fault
moveUser	Valid	Pass	
moveUser	Invalid (move a user that does not exist)	Pass	
removeUser	Valid	Pass	
removeUser	Invalid (remove a user that does not exist)	Pass	
endSubConf	Valid	Pass	
endSubConf	Invalid (end a non-existent conference)	Pass	The observer detects an INPUT fault
endConf	Valid	Pass	
endConf	Invalid (end a non-existent conference)	Pass	The observer detects an INPUT fault

The "initiateConf" fails twice during observation. The first fault occurs when it's invoked with an empty list of initial users, no response is received. The second fault occurs when giving a conference type different than "dial out" (dial in for example) which has no effect and a dial out conference is initiated in all cases. The first fault cannot be detected by the FSM-based observer since the request was valid and no response is received even when waiting for a long time. The second fault, to the contrary, is detected. In the scenarios when the observer detects an INPUT fault, the FSM machine was in the "Idle" state. At this state, the client can only initiate a conference. An INPUT fault states that the client requests an operation from an invalid state. The Web Service passes all tests when an INPUT fault was observed. After observing the Call Control Web Service, it seems that the Web Service, and the Parlay gateway behind, is more or less well tested (with regards to the executed scenarios) except for the two cases for which the Web Service fails.

6.2.4 Quantitative evaluation of the architecture

Our management architecture can be quantitatively evaluated using three criteria: pre-observation configurations, processing CPU and memory utilization, and generated network load.

6.2.4.1 Pre-observation configurations

The traces' collection mechanisms presented in section 6.1 are more or less easily configurable. To instrument the client (or the Web Service), 4 basic java statements, in a try/catch block have to be added to the code to initialize the observer. Moreover, to forward each trace, 2 java statements, also in a try/catch bloc, are appended to the source code.

The TCP-dispatcher approach requires a light modification to the WSDL document of the Web Service. The IP address/host name and the port number of the port tag (wsdl:port tag in Figure 5.2) in this document should be replaced by the IP address and port number on which the dispatcher is listening. The TCP-dispatcher is then configured as shown in Figure 6.8:

- **Listen port #**: this is the port on which the dispatcher is listening for requests from the client. It should be the same as the port number in the updated WSDL document.
- **Web Service options**: these are the IP address/host name and port number of the Web Service; that is, the initial value from the WSDL document.
- **Observer option**: the IP address/hostname and port number of the Web Service Observer. This information can be obtained from the WSDL document of the WSO.

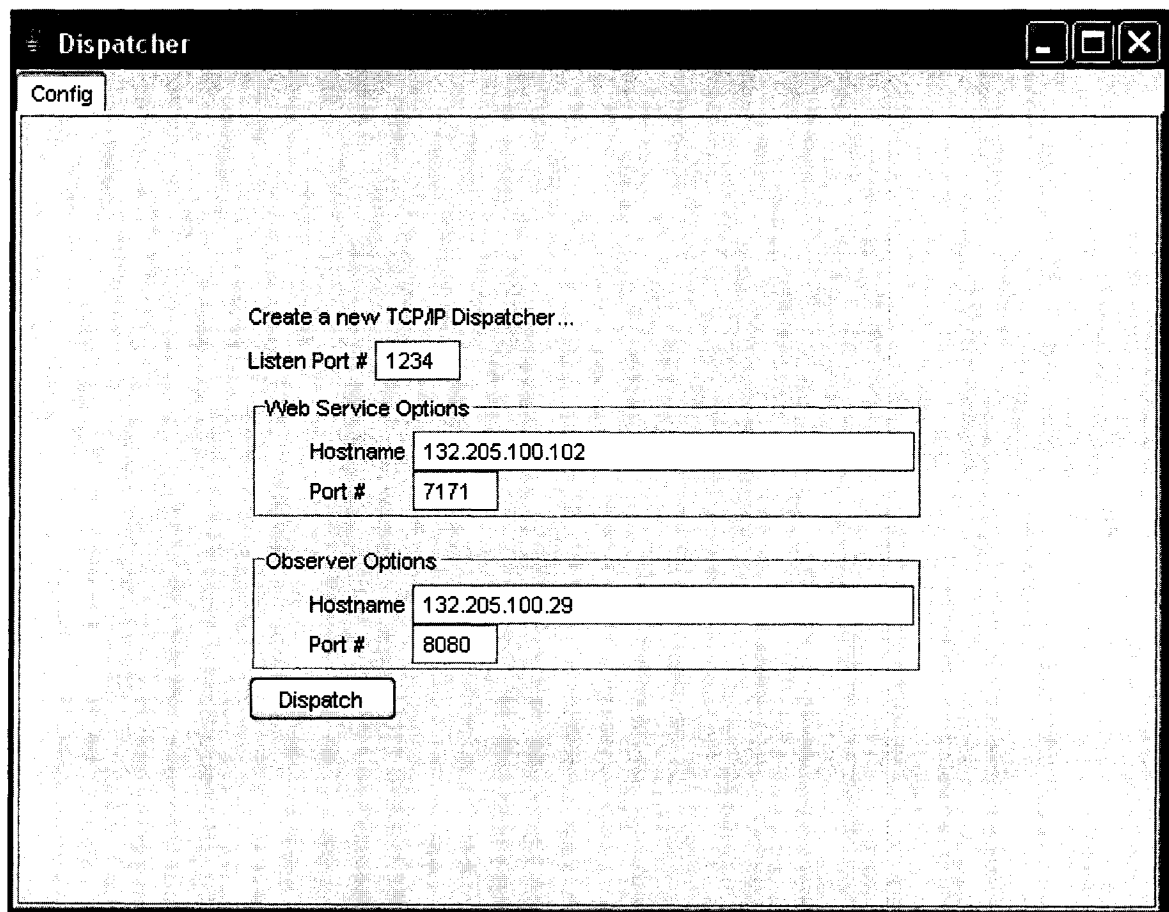

Figure 6.8 TCP-dispatcher configuration

Once configuration is finished, pressing the "Dispatch" button activates the dispatching operations as illustrated in Figure 6.9.

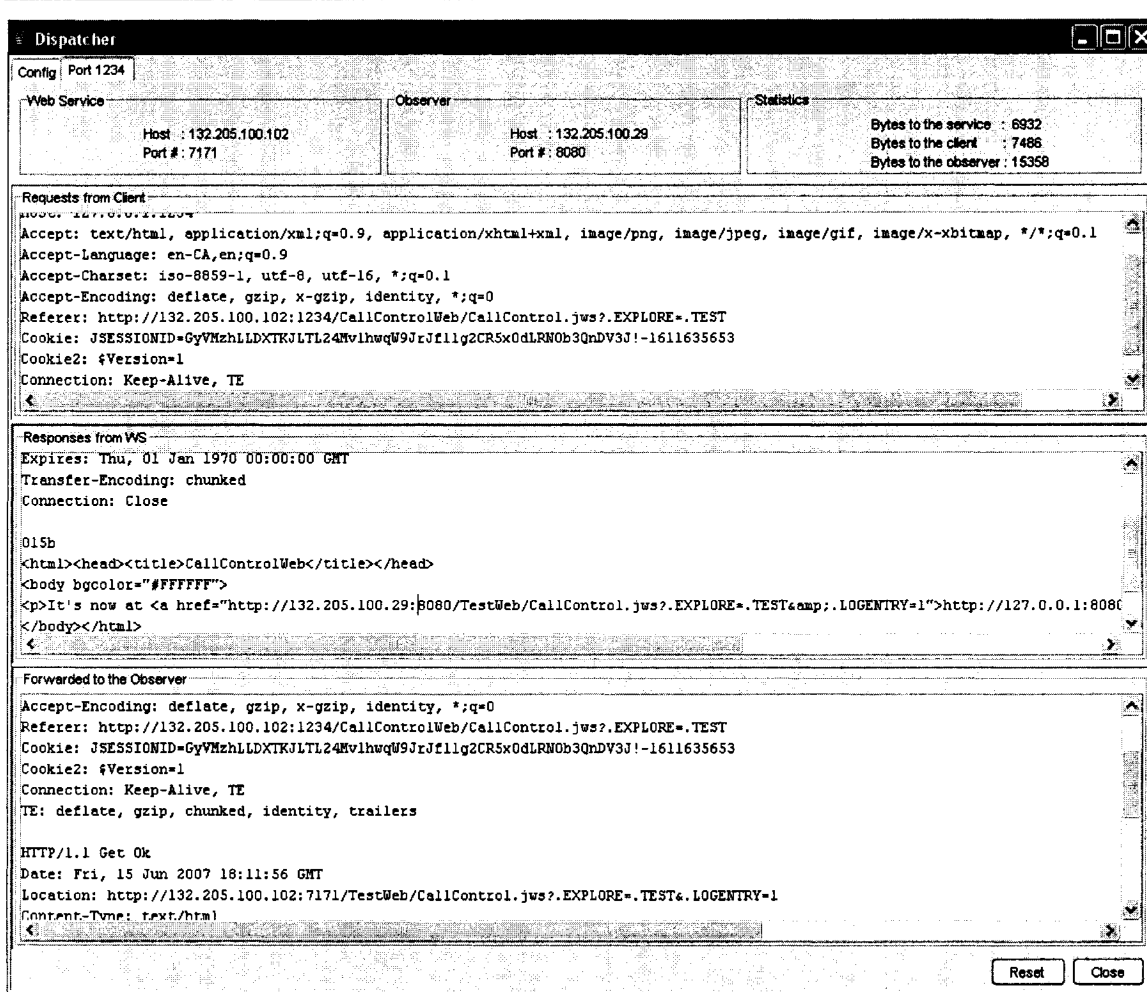

Figure 6.9 TCP-based dispatching

When using SNMP for traces' collection, a complete MIB and an SNMP agent are required. The agent generates traps from a log file (MIB) and sends them to the Web Service Observer (Figure 6.10). The log file should be maintained by the client or the Web Service and the agent must be running on a node having access to this file.

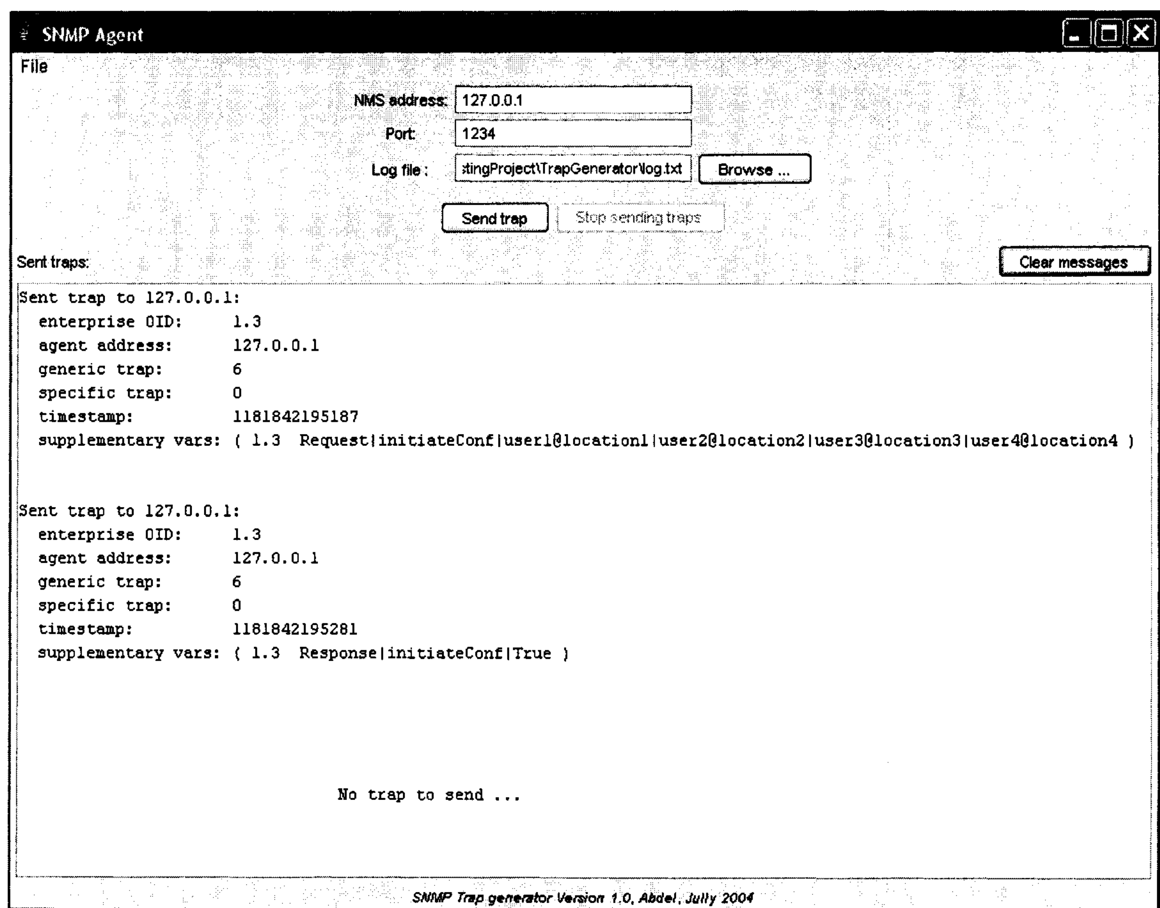

Figure 6.10 SNMP agent traces' collector

Designing the observer as a mobile agent requires a flexible mobile agent platform. The Java Agent Development Framework [101] has been used. In addition to its open source feature, its flexibility, ease of configuration and use justify its utilization in the design of our mobile observers. The platform consists of a set of libraries (8 Java archive -jar- files) that can be downloaded offline and executed. A mobile observer is illustrated in Figure 6.11 where the GUI shows received requests with their invocations parameters' names, types, and values (upper rows) and the output type of the corresponding response (lower rows). The verdict of the observation is shown in the status bar (bottom of the window).

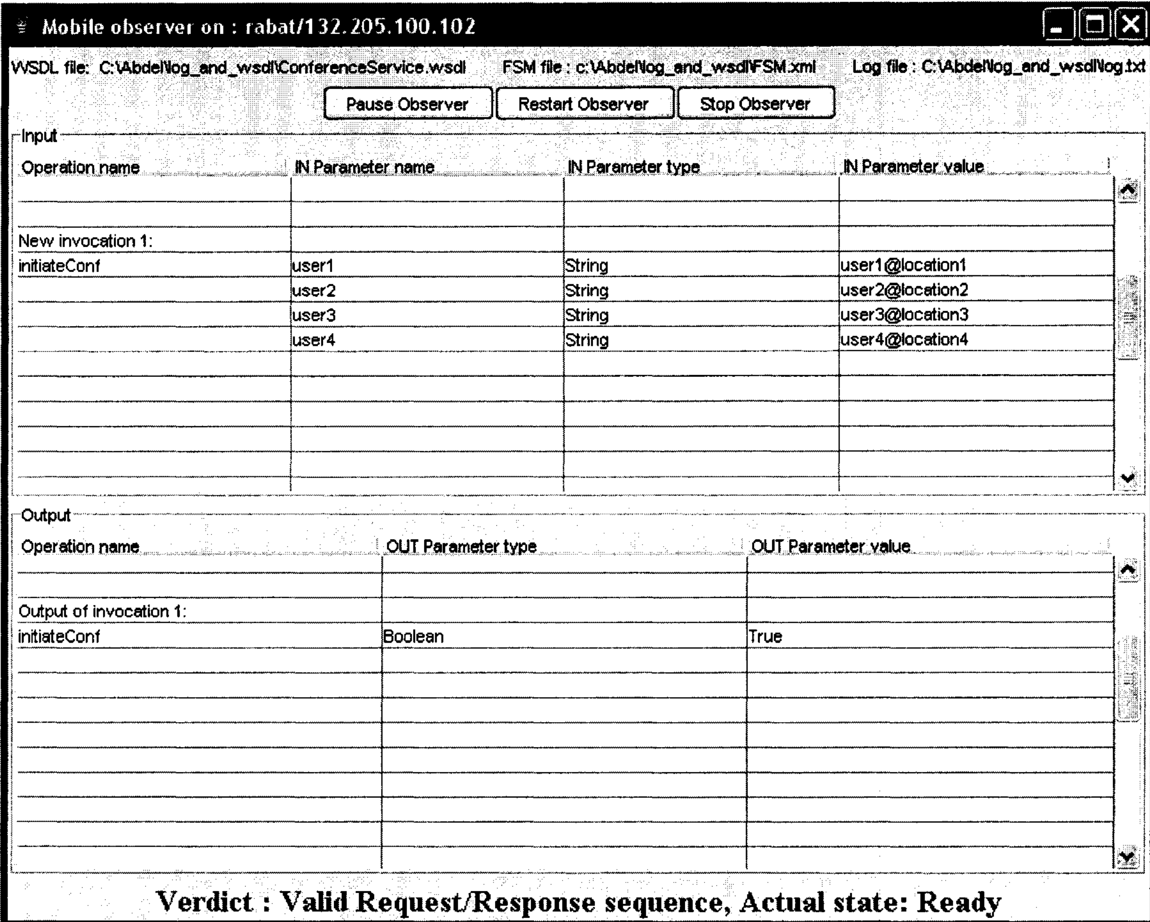

Figure 6.11 Mobile observer GUI

6.2.4.2 Processing CPU and memory utilization

Computer resources used by traces' collection entities are somehow insignificant with regards to the minimal standard configuration of actual personal desktops and laptops. Except for the mobile agent approach, CPU and memory utilization are so low that they are even difficult to evaluate.

For mobile observers, CPU and memory utilization on a laptop equipped with an AMD Athlon 64/3000+ processor and 512MB RAM, CPU and memory utilization are as follows:

- **Hosting a mobile platform:** if the mobile agent administration interface is located on the laptop, the CPU usage varies between 2% and 4%. For memory, it uses around 30 Megabytes.
- **Joining a mobile platform:** if the mobile agent platform is running on a remote computer, joining it requires 12 MBytes memory at the laptop and around 2 MBytes on the host running the administration interface. For CPU, there is almost no impact at both sides. A node can host a mobile observer if it is running a mobile agent platform administration interface or is joining a running remote platform.
- **Receiving a mobile observer:** when a mobile observer is received, it requires around 27 MBytes of memory. For CPU, there is a high utilization during 1 to 2 seconds while initializing and displaying the graphical interface of the mobile observer, then the CPU utilization goes back to previous level.
- **Processing traces:** even in the worst case, where traces are received with a very small delay, the CPU used by the mobile observer for analyzing them is around 2%. However, there is no additional memory utilization.

6.2.4.3 Network overhead

For each request or response, a message is generated to the observer by invoking respectively "sendRequestToObserver" or "sendResponseToObserver". Except when using mobile observers, passive observation doubles the traffic on the network due to traces' collection. The two interfaces have one parameter each which is the event (request or response). Due to its structure, a SOAP message carrying one parameter has almost the same size (from

a complexity point of view) as a SOAP message carrying two parameters. Combining the two traces (a request and its response) using "sendRequestResponseToObserver" reduces the overhead by almost 50%.

For client instrumentation and the dispatcher, each forwarded SOAP message has a size of 2 Kbytes. If n denotes the number of requests/responses pairs, the overhead using "sendRequestToObserver" and "sendResponseToObserver" is around $2n$ Kbytes. However, the overhead is n Kbytes if the trace collection entity uses "sendRequestResponseToObserver".

In the SNMP mechanism, the size of each packet is between 150 and 200 bytes. If each entry in the MIB (request or response) is sent in a separate packet, the overhead is between $0.15n$ and $0.2n$ Kbytes and half of it if using "sendRequestResponseToObserver".

The overhead associated with the mobile code mechanism is generated by moving the mobile observer to the client or Web Service side. This operation generates a 600 Kbytes overhead: the size of the mobile observer.

Figure 6.12 illustrates measures of the network load introduced by the observation using the communication mechanisms presented above. Even with more or less big size of the mobile agent, the mobile agent approach presents the lowest network overhead. This is due to the fact that the observation traffic over the network is constant and independent from n.

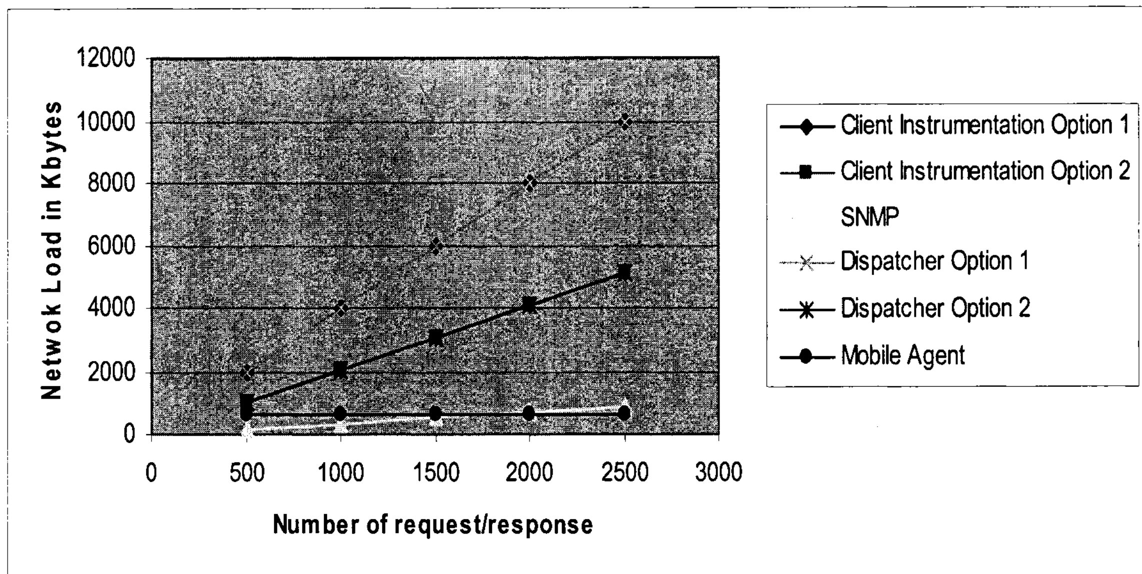

Figure 6.12 Network load measures for traces' collection mechanisms

If the number of request/response pairs is less than 2000 request/response, only the SNMP mechanism generates less or similar overhead as the mobile code approach. For request/response pairs exceeding 2000, the load generated by the SNMP mechanism continues to grow while the load introduced by the mobile code mechanism remains constant.

6.2.5 Discussion

Comparison of traces' collection mechanisms discussed in previous sub-sections can be based on 5 criteria: CPU and memory utilization, participation of client and/or Web Service's provider, required changes to client and/or Web Service source code, generated network load, and possible introduced delay. The first criteria can be neglected since CPU and memory required resources are somehow low for all mechanisms. Remaining criteria are shown in Table 6.3:

Table 6.3. Characteristics of Traces' collection mechanisms

Criteria / Mechanism	Client/Web Service's provider participation	Changes required to client/service	Network Load	Time Delay
Client Instrumentation	High	High	High	None
SNMP	Low or None	Low or None	Medium	None
Dispatcher	Low	None	High	Low
Mobile observer	Low	None	Low	None

The mobile observer approach seems to be a good strategy for passive observation. First of all, participation of the client/Web Service's provider is limited to hosting a mobile platform. The latter can be downloaded offline and easily configured. Second, it generates the lowest network load for more than 2000 pairs of request/response. Since the number of request/response pairs for a Web Service will be higher than 2000, we can conclude that mobile agents, whenever their utilization is possible, are the most suitable mechanism for communication between different components of the observation architecture. For this reason, mobile observers will be used in the remaining chapters of this thesis.

6.3 Summary

In this chapter, we studied potential traces' collection mechanisms. We proposed and compared different approaches through a real case study. This comparison leads to the consideration of mobile observers as the best mechanism for traces' collection. This experimenta-

tion allowed also an evaluation of the effectiveness of the management architecture in terms of misbehaviors' detection.

The study performed in this chapter covers all aspects for passive observation of a basic Web Service. However, it does not support management of composed Web Services. This will be thoroughly studied in the next chapter.

Chapter 7

Observation Architectures for Composite Web Services

The whole is often more than the sum of its parts -
Aristotle

Web Services can be developed following two directions: built from scratch or based on composition using already available Web Services. The second approach aggregates existing Web Services to create a more complex Web Service providing a richer range of functionalities. As discussed in section 2.2, few description languages are used for Web Services composition. Among them, BPEL is gaining a lot of interest and becoming a standard for composition of Web Services.

Management of composite Web Services may require a multi-observer architecture as discussed in section 4.1.3 and illustrated in Figure 4.3. This type of architectures offers to manage, in addition to the composite Web Service, the basic Web Services participating in the composition.

The study and experimentation conducted and exposed in the previous chapter showed that mobile observers generate the lowest overhead. When managing a composite Web Service and its basic Web Services, the number of observers and location of points of observation affect the effectiveness of the architectures in terms of misbehaviour detection and network load. Locations of observers have an impact on the generated overhead. For example,

* Results from this chapter have been published in [86], [87], [88], [89], [102], and [103].

network load is different if all observers are located within the same network or scattered at many locations.

This chapter is dedicated to management of composite Web Services. The chapter is divided into four sections: the following section presents the required procedure to make use of the management architectures. The second section studies different locations where mobile observers can be hosted and how these locations affect its effectiveness. It gives also a set of heuristics to decide on the number of observers. Section 7.3 presents examples of algorithms that observers must implement. Section 7.4 uses a case study of a composite Web Service to assess the multi-observer architectures. Some QoWS aspects will be observed in addition to functional behaviour.

7.1 Procedure

Observation of composite Web Services is performed in two main steps. The first step consists of the configuration of observation components (global and local observers), and the second step is misbehaviours' detection.

Observation is initiated by invocation of the WSO. After a successful invocation, the WSO generates a set of mobile agents and sends them to the location(s) specified during invocation. The number of generated mobile observers depends on a set of parameters as will be discussed in section 7.2. Once the mobile observers reach their target locations, one of them becomes the global observer, others are local observers.

The local observers must inform the global observer of their locations and which Web Services they are observing. At that point, all the components of the architecture are ready to start observation at the time specified during the invocation. This observation will last at the

time specified, also, during invocation. Whenever misbehaviours are observed, local observer(s) report to the global observer who reports to the WSO.

7.2 Number and locations of mobile observers

Observing composite Web Services necessitates additional information compared to observation of simple Web Services. First of all, the global observer must have access to the BPEL document of the composite Web Service. Second, since observation of composite Web Services requires also the observation of all or a subset of basic Web Services, the list of these should be handed to the WSO at invocation.

One of the design keys to be studied is the number of observers. In some cases, observing all basic Web Services might be costly and useless. A complete observation might generate redundant information which is overwhelming observers and network. In partial observation, the observation of a sub-set of Web Services, for example those that represent the core of the composition can be sufficient from misbehaviours' detection point of view. However, in partial observation, not all misbehaviours can be detected since some of them can be tolerated or even compensated by interacting Web Services. Thus, before initiating observation, the list of Web Services to observe is built.

An important criterion in selecting Web Services to observe is the number of interactions between the composite Web Service and a specific basic Web Service. This information can be drawn from the BPEL document. If a Web Service has few published interfaces and is invoked few times while others are invoked very often, observing the latter Web Services can be more appropriate than observing others. Another criterion is the complexity of a basic Web Service which can be derived from its behaviour's specification. More behaviour's

specification of a Web Service is complex (number of states, number of transitions ...), higher is (or might be) the necessity for its observation.

Statistics on previous detected misbehaviours is another criterion. If faults occurred in a Web Service a certain number of times, a periodic observation of this Web Service might be a wise decision.

Selection of Web Services to observe can be implied by preferences of the composite Web Service's provider. These preferences might depend on the importance a basic Web Service is playing in the composition and/or the tolerance of the composite Web Service to some specific faults generated by some specific Web Services. The preferences can be triggered also after modification and/or maintenance operations performed on one of the basic Web Services or the composite Web Service itself.

Generally, observation of composite Web Services requires then the following information and resources:

- List of Web Services to observe
- Behaviours' specifications of these Web Services (BPEL, FSM, FSM+, EFSM...)
- Their WSDL documents
- Hosts of mobile observers

The information and resources required for observation can be gathered through participation of involved Web Services' providers: the composite Web Service provider, providers of basic Web Services, or from both. We designate these types of participation, respectively, as *composite Web Service provider's participation*, *basic Web Services providers' participation*, and *hybrid participation*.

These three types of participations imply three variants of the multi-observer architecture as will be shown in sections 7.2.1, 7.2.2, and 7.2.3.

7.2.1 Composite Web Service provider's participation

In this participation, the provider of the composite Web Service provides all required information and resources necessary for the observation. This kind of participation is completely transparent to basic Web Services and their providers. All observation activities are performed within the composite Web Service provider's side and basic Web Services' providers never notice that such observation is actually taking place. All mobile observers (local and global) are located in the same domain or even the same node as illustrated by Figure 7.1.

Moreover, this participation has a multitude of strengths. Due to the cloning nature of mobile agents, the WSO sends only one mobile observer to the composite Web Service provider's side instead of a separate mobile observer for each Web Service to be observed. This mobile observer will clone itself once it gets into its hosting location. Doing so reduces significantly the traffic generated by moving mobile observers. If m is the number of Web Services to be observed, the complexity of the introduced load goes down from $\Theta(m)$ to $\Theta(1)$.

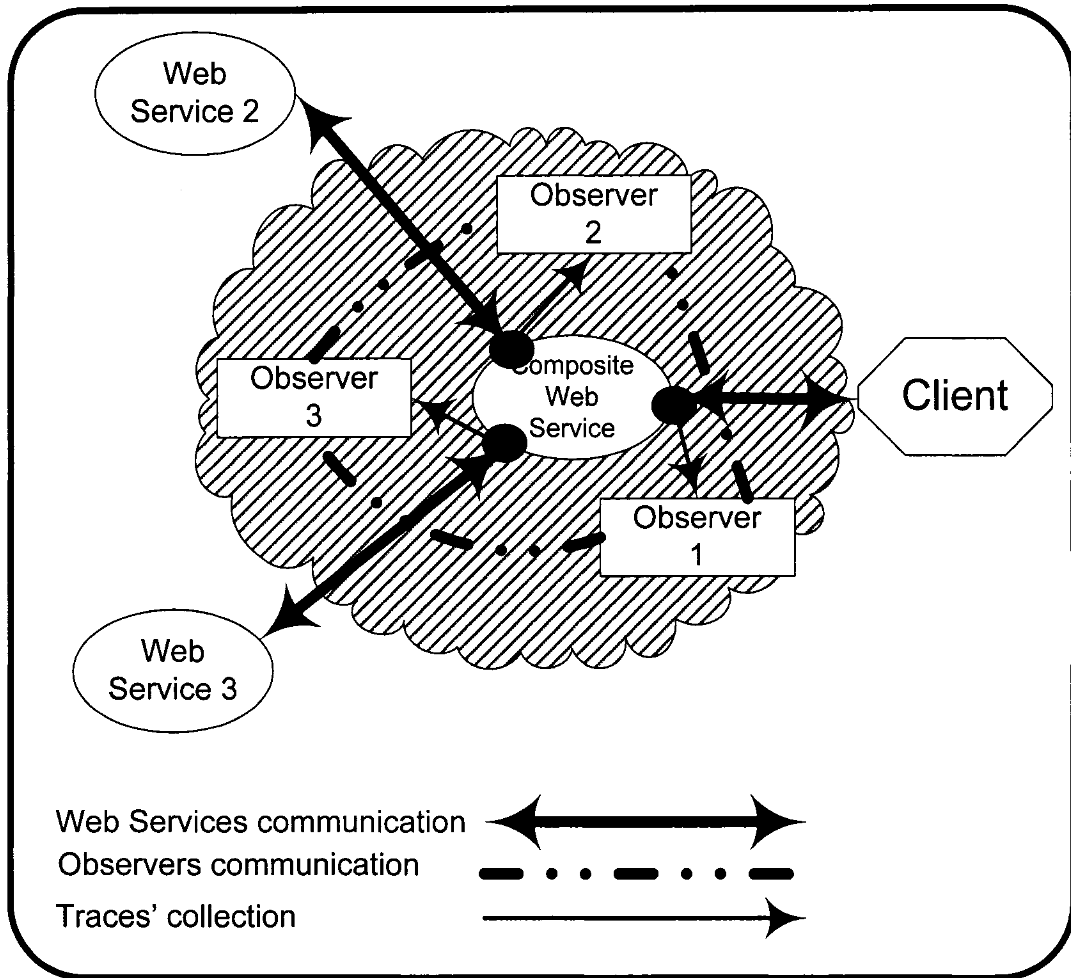

Figure 7.1 Composite Web Service's provider participation

The network load that will be introduced by the cooperation of observers to detect and locate misbehaviour is limited to in-site load, that is, within the provider's domain where all observers are located. Nowadays, local networks (e.g. switched Ethernet) offer huge bandwidth and the complexity of this load can be considered as $\Theta(1)$. Synchronization of observers is also easier than if observers were scattered between many sites.

The weakness of the composite Web Service's provider participation is that all information and resources should be within one Web Service provider. This provider also hosts all

the mobile observers, thus requiring more resources. This weakness can be ignored due to the limited resources a mobile observer is consuming (see sections 6.2.4.2 and6.2.4.3), except the need for a mobile platform which is required regardless of the number of mobile observers.

7.2.2 Basic Web Services providers' participation

Unlike the centralized participation presented in the previous sub-section, the basic Web Services' participation requires the participation of all providers of Web Services that have to be observed, including the composite Web Service. Each provider supplies the WSDL document and the behavior specification document of its Web Service and hosts the associated mobile observer (Figure 7.2).

The monitoring activities are distributed among all observed Web Services as represented by the dashed areas in Figure 7.2. This distribution reduces the needed resources for the observation between Web Services' providers rather than centralizing them in one side.

The network load, in terms of extra packets, is the major weakness of this type of participation. First, a mobile observer is generated and sent to each Web Service in the list of Web Services to be observed. The complexity of the load in this is $\Theta(m)$. The cooperation of the observers introduces also another $\Theta(m)$ network load since observers are in different locations.

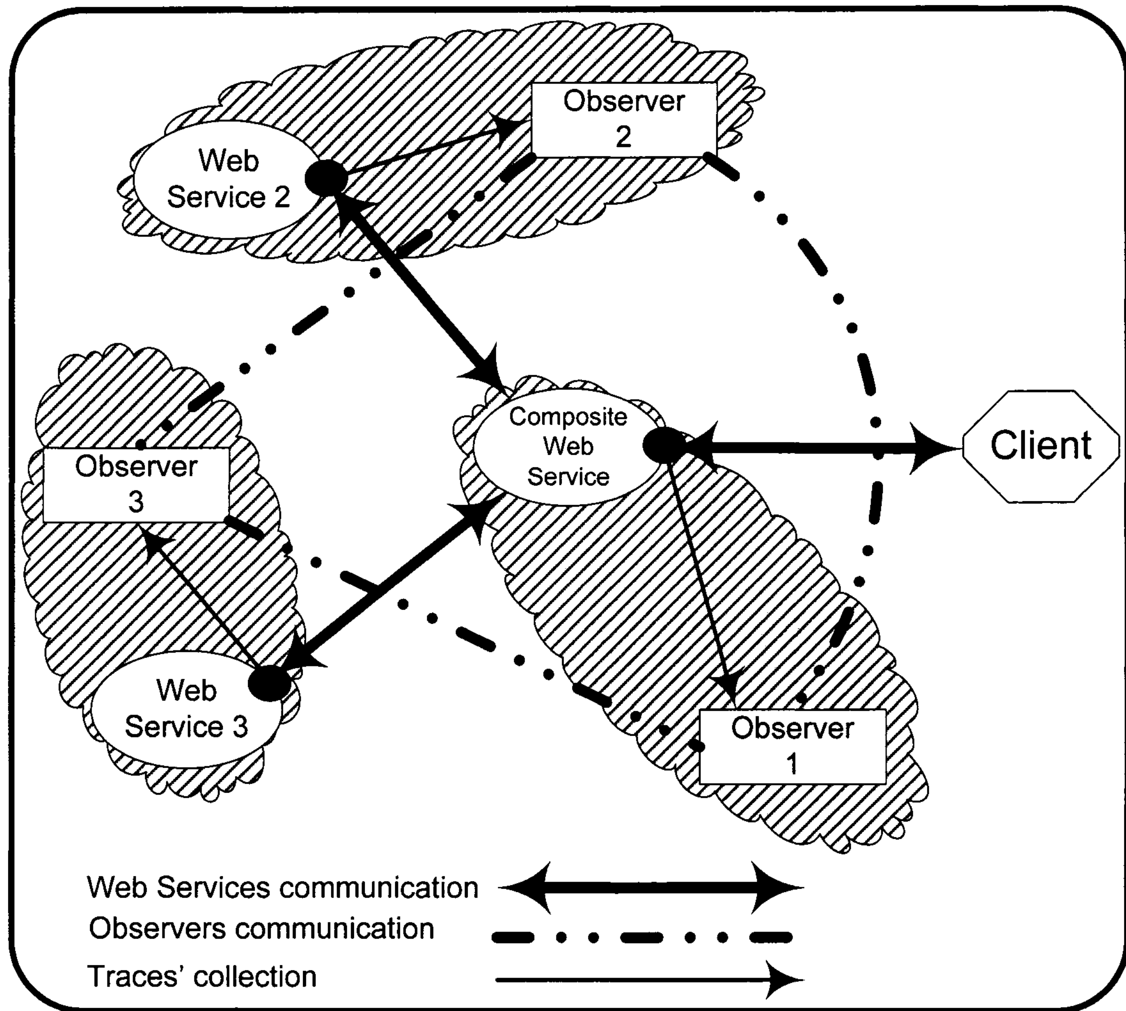

Figure 7.2 All providers' participation

7.2.3 Hybrid participation

The hybrid participation is a compromise between the two kinds of participation presented above. The participation is neither completely distributed over many sites nor centralized in one site. The composite Web Service provider supplies a portion of the required information

and resources while a subset of the list of basic Web Services to be observed supplies remaining portions (Figure 7.3).

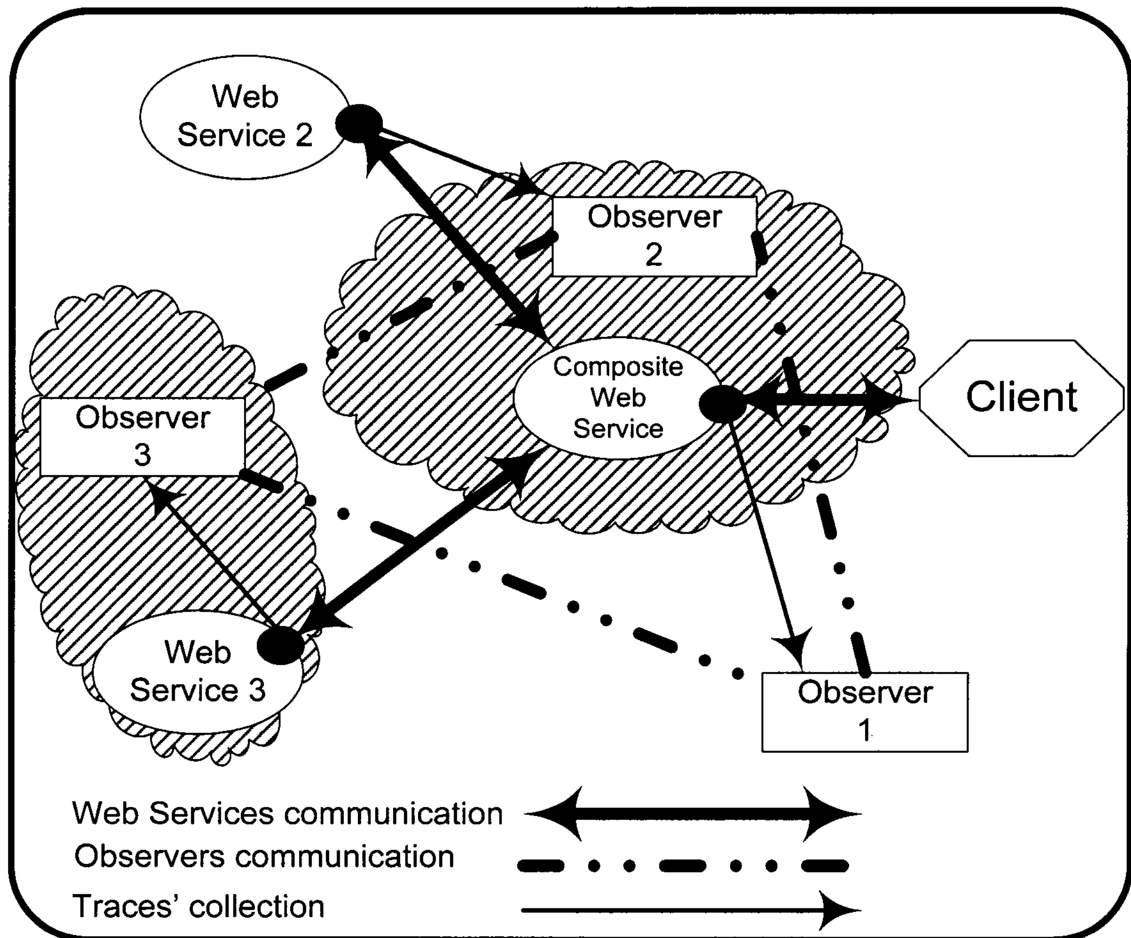

Figure 7.3 Hybrid participation

The above architecture can be a possible alternative when the provider of the composite Web Service cannot provide all the information and resources and only a subset of basic Web Services' providers is willing to participate in the observation. Those basic Web Services providers' who accept to participate in the observation will supply information related

to their Web Services and host associated mobile observers. The provider of composite Web Service supplies information for other basic Web Services.

The configuration of the hybrid participation ranges between the centralized and the distributed architectures, depending on how many basic Web Services providers' are participating in the observation and how much. Thus, the complexity of the load generated by moving the mobile observers ranges from $\Theta(1)$ to $\Theta(m)$ and for the cooperation of observers from in-site load to $\Theta(m)$. In the average, these complexities are around $\Theta(\log m)$.

7.2.4 Discussion

Architectures presented in 7.2.1, 7.2.2, and 7.2.3 can be compared with regards to misbehaviors' detection capabilities and generated network overhead.

In terms of misbehaviors' detection, the three architectures presented above are equivalent if traces' collection mechanisms use a reliable communication mode (TCP or acknowledgments for example). This equivalence is due to the fact that the points of observation are unrelated to the location of observers (Figure 2.6, Figure 7.1, Figure 7.2, and Figure 7.3). In such configuration, an observer of a specific Web Service has access to the same traces wherever it is located. In Figure 7.3 for example, even if observer 2 is located at the composite Web Service's provider, it gets the trace collected at the provider of Web Service 2. Moreover, this trace is checked against the model of Web Service 2 no matter where it is collected and analyzed.

Misbehaviors' detection can be impaired when using a non-reliable communication mechanism for traces' collection. If a trace carrying a faulty interaction is lost and cannot be

recovered, the fault in this interaction will never be detected. In environments where lost packet cannot be recovered (by forward error correction or retransmission for example), the architecture in 7.2.1 is suitable. An approach where sequence numbers are used to detect message loss is presented in section 7.4.

For network overhead, composite Web Service's provider participation has the lowest, basic Web Services' providers participation has the worst, and the hybrid participation ranges between them. This is basically implied by the number of mobile observers and traces that have to transit over the network to their final destinations. More experimental network overhead analysis is provided in section 7.4.6.

The next section gives examples of algorithms that must be implemented in observers. These algorithms will be used in the case study of section 7.4.

7.3 Examples of Observers' algorithms[3]

An overview of the algorithm for misbehavior detection is depicted in Algorithm 7.1. Both global and local observers must implement this algorithm. Every observed event (request or response) is checked against the expected behavior. Whenever a trace is not valid, the fault should be purged and/or correlated to previous notifications. Otherwise, a fault notification is generated.

[3] Algorithms presented in this section are not exhaustive. They have been designed for the case study of section 7.4. Further development might be required for other situations.

```
Input : event e
Data : I Set of Input
Data : O Set of Output

// Event does not exist
if (e ∉ (I U O)) then
 └ return "UIF Fault detected"
// event is a Request
if (e ∈ I) then
|   if (e not expected) then
|    └ return "UIF Fault detected"
|   if (signature of e is invalid) then
└    └ return "ITF Fault detected"
else
|   // event is a Response
|   if (e ∈ O) then
|   |   if (e not expected) then
|   |    └ return "UOF Fault detected"
|   |   if (return type of e is invalid) then
|   |    └ return "OTF Fault detected"
|   |   if (RT > MxRT) then
|   |    └ return "MxRT Fault"
|   |   if (RT < MnRT) then
└   └    └ return "MnRT Fault"
return "Valid trace" /* if no fault detected before, the trace
   is then valid                                          */
```

Algorithm 7.1 Misbehavior detection

Whenever a fault is detected by a local observer, a notification is sent to the global observer. Notifications must be purged before correlation. This is done through two methods: purgeFinalNotification implemented by the global observer (Algorithm 7.2) and purgeLocalNotification implemented by local observers (Algorithm 7.3).

```
Input : fault f
Data :  Fault Record (FR)

if (f ∈ (UIF, TF, ITF)) then
    wait for a response from the composite WS
    if (fault detected by composite WS) then
        add f to FR
    else
        correlate(f)
else
    if (f is a OTF) then
        wait for the reaction of the client
        if (fault detected by client) then
            add f to FR
        else
            correlate(f)
    else
        // f is a RT fault
        correlate(f)
```

Algorithm 7.2 Global purge

The main purging role is the ability of a receiver (client, composite Web Service, or basic Web Service) to detect a faulty received request or response. When a local observer detects an output fault, it notifies the global observer. It waits then for the reaction of the receiving entity. If the response of the latter contains a fault indication (in the SOAP message), the local observer informs the global observer. Otherwise, it sends a second notification to the global observer requesting fault correlation and location (Algorithm 7.4).

```
Input : fault f
Data :  Fault Record (FR)

// inform global observer about the fault
notifyGO(ObserverID, f)
if (f ∈ (UIF, TF, ITF)) then
    wait for a response from the WS
    if (fault detected by the WS) then
        /* inform global observer that the WS detected the
           fault                                                    */
        notifyGO(ObserverID, "last Fault detected by WS")
        add f to FR
    else
        notifyGO(ObserverID, "last fault pending for solve/correlation")
else
    // the fault is a OTF
    wait for the reaction of the composite WS
    if (fault detected by composite WS) then
        /* inform global observer that the composite WS
           detected the fault                                       */
        notifyGO(ObserverID, "last Fault detected by composite WS")
        add f to FR
    else
        notifyGO(ObserverID, "last fault pending for solve/correlation")
```

Algorithm 7.3 Local notification and purge

```
Input : fault f
Data : Fault Record (FR)
Data : Resolved Faults RF

if (f can be associated to an element of FR) then
    update FR
    notify the Web Service Observer
    return "fault correlated"
wait for a 2nd notification // Fault pending or not from LO
if (f is pending) then
    update FR
    if (f ∈ RF) then
        listOfSuspects = associatedFaultyServices // previous f from
            FR
    else
        listOfSuspects = preceeding WS in BPEL
    repeat
        check traces and notifications regarding all WS in listOfSuspects
        remove items from listOfSuspects whenever possible
    until ((|listOfSuspects| == 1) || (remaining WS ∈ listOfSuspects are
    not Observed) || (no decision can be made))
    update RF
    update FR
    notify the Web Service Observer
```

Algorithm 7.4 Correlation

A faulty response generated by a Web Service will be detected as an output fault by its associated observer. It can also be detected as an input fault by the observer attached to the receiving Web Service unless the latter is fault-tolerant. Both observers will generate fault notification. The two notifications must be correlated since they refer to the same fault.

After receiving a notification from a local observer, the global observer associates it, if possible, to a previous fault or notification and updates the fault records accordingly. It waits

then for a second notification for a specific period of time before initiating the search for possible correlations. This starts by checking the fault records for previously detected fault. If the same fault has been detected before, the list of suspected Web Services is updated with the faulty Web Service(s) in the fault record. The list of suspects is then augmented by all basic Web Services invoked before the notification. This list is derived from the "activities" section of the BPEL document. Traces observed by the local observers of the Web Services in this list are checked to find the faulty Web Service. This process is repeated until a faulty Web Service is identified, remaining Web Services are not observed, or no decision can be made due to a lack of information on behaviors.

In the next section, we illustrate the applicability and the effectiveness of the multi-observer architecture through a motivating example of a composite Web Service for conferencing. The detailed requirements and steps for observation are depicted all along this example.

7.4 Case study

In this section, we present our experiments using multi-observer architectures to observe a composite Web Service. We experimented the three multi-observer architectures proposed above; but in this section, we show scenarios from the experimentation of the hybrid Web Service's provider participation. However, a comparison of the network overhead caused by the three architectures is given.

This section first introduces the case study, a composite Web Service and its basic Web Services. Then it shows a situation where the observation of basic Web Services gives more insights for misbehavior detection and identification. Finally, it presents implementations of

different components of the architecture and discusses results of experimentation with analysis.

7.4.1 Case study description

For the end of year meetings, a general manager has to meet with managers from different departments (e.g. Sales, R&D...). Managers are located in different locations and due to their busy timetables, they cannot meet in a single location. A practical alternative is to perform these meetings in a series of teleconferences. Only managers are concerned and only those of them that are in their offices can join a conference. This is implied by security issues since confidential information will be exchanged during the meetings and communication between different locations is secured (Virtual Private Network, VPN, for example). At the end of each meeting, meetings' reports must be printed and distributed among all participants.

The manager decides to use a "Conferencing Web Service" (CWS), a composite Web Service, who performs all of the required tasks. In fact, it allows creation of conferences, add and remove participants to conferences depending on their profiles and physical locations. At the end of each meeting, the CWS submits the produced reports for printing. Once printed and finalized, the paper version is distributed to appropriate locations.

The general manager is highly concerned with the environment in which meetings will be carried out using CWS. He decides to make use of the management architecture to assess the behavior of the CWS.

7.4.2 Web Services

To perform all these tasks, the CWS is a composition of the following basic Web Services:

- Presence: this Web Service contains information on users' profiles (name, address, location, status, position, availability).
- Sensors: this Web Service detects the physical presence of users.
- Call Control: this Web Service creates and manages a multiparty conference (initiates the conference, adds/removes participants, and ends conferences).
- Printing: at some points during the conferences or later on, managers may want to print documents (meeting reports ...). The printing Web Service will print these documents and keeps them for shipping.
- Shipping: documents printed during and after the conference should be distributed among participants located in different locations. The CWS informs the shipping Web Service of the location of the documents to be shipped and their final destinations.

Figure 7.4 shows the composite CWS and its interactions with the basic Web Services.

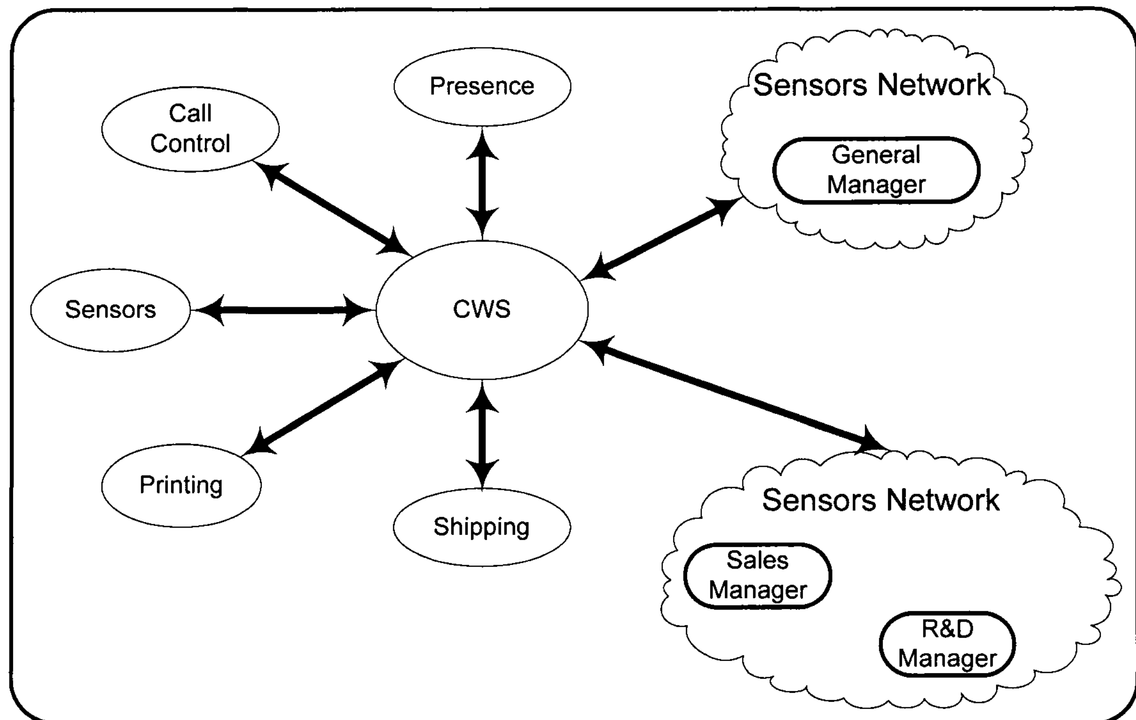

Figure 7.4 Composite/basic Web Services

7.4.3 Implementation issues

All Web Services, including the WSO, are implemented in BEA WebLogic. In fact, CWS is implemented in BEA even if it has a BPEL description. This is due to some limitations of the BPEL language and the available (non-commercial) application servers. Implementing the CWS in BEA does not affect the observation process since the latter deals only with the exchanged SOAP messages which are independent from the adopted platform (BPEL Engine or BEA).

Mobile observers get traces using SOAP Handlers available within the BEA platform. A SOAP Handler, a specific-purpose Java class, intercepts a request/response to/from a Web

Service before it gets to the core Web Service or the client respectively, and can also perform operations on it. In our case, the SOAP handler sends each intercepted request or response in a User Datagram Protocol (UDP) datagram to the concerned mobile observer.

The date of occurrence of each event is also sent in the corresponding datagram so that the observer can assess QoWS attributes. Appending occurrence date to each event raises a much known and complicated issue of synchronization of distributed systems. In this thesis, I do not propose new solutions to this synchronization matter; I however, use two available and widely used mechanisms:

1. Network Time Protocol (NTP): deploying and configuring an NTP in a local network where observers are located seems to solve the problem. Nevertheless, when mobile observers are located at different locations, this solution might show limitations.
2. Mobile platform clock: mobile observers/agents can request an accurate clock value from their hosting platform. Here gain, if locations of mobile observers are quite far from each other, sharp accuracy of such synchronization cannot be guaranteed.

The mobile observer checks each received trace and forwards it to the global observer, also in a UDP packet. Since the behavior/operation of SOAP handlers within all observed Web Services is similar, a unique (generic) SOAP Handler is developed and then distributed to all providers participating in the observation.

Although UDP is an unreliable protocol, lost of UDP datagrams is not very frequent when the sender and the receiver are located on the same local network (this is the case of

the composite Web Service participation). In fact, the data link layer supporting UDP has its own flow control and recovery mechanisms. However, when communicating entities resides on different networks (this is the case with the basic Web Services' providers and hybrid participation), a UDP packet might have to go through many public networks where probability of packet loss is somehow high.

To be able to detect lost UDP datagrams, a sequence number field is used. When a mobile observer detects a lost datagram (wrong/not expected sequence number), it suspends the misbehavior detection and re-perform the homing procedure. After all, this is another situation where an observer does not have access to the full set of traces. It restarts the detection once this procedure is achieved correctly.

7.4.4 Single observation limitations

When using the single-observer architecture presented in 4.1.3, the observer will check only the traffic between the manager and the CWS. Figure 7.5 shows the overall configuration and the information (traces) available to the observer where it is not aware of the interactions (request/response pairs) between CWS and basic Web Services. By doing so, if the CWS fails to provide the requested service or if the QoWS degrades, the observer might not designate the faulty Web Service. For example, if the "Sensors" Web Service (Basic WS) fails to check the actual physical location of a manager, the CWS cannot add a manager to a conference. From the observer's point of view (and then the manager's point of view), the CWS failed to add the manager to the conference. No more indication on the failure is available.

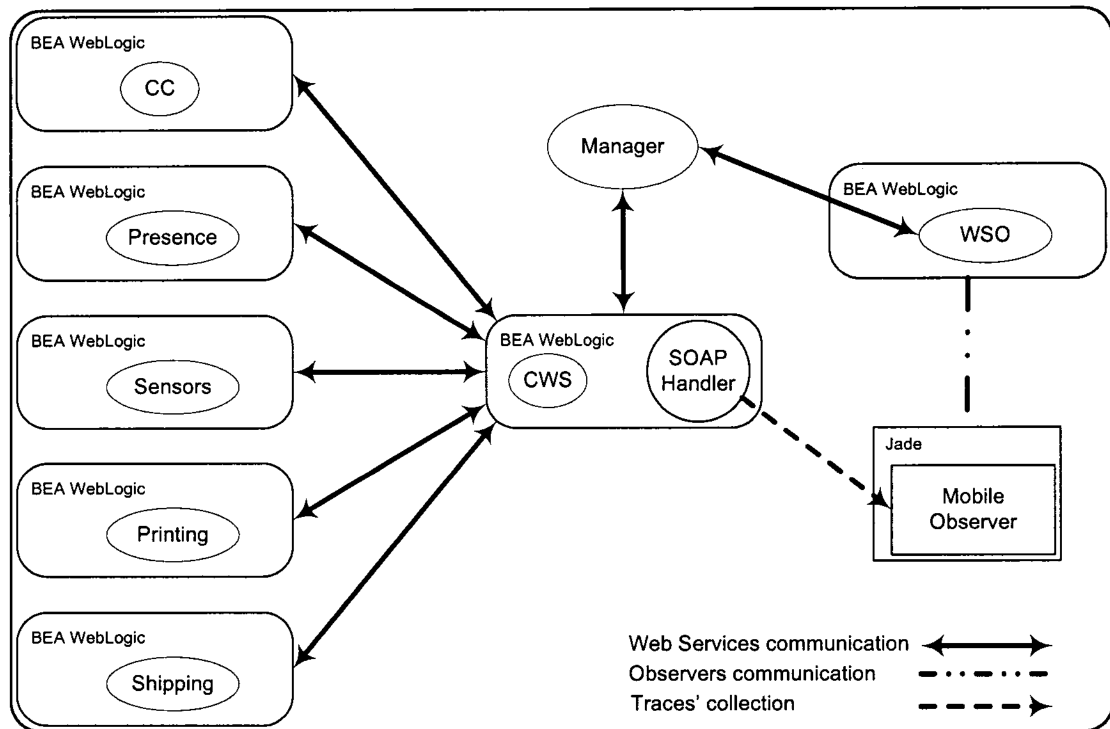

Figure 7.5 Single-observer configuration

Figure 7.6 shows a typical observation scenario from invocation of the observer (WSO) to the delivery of the verdict of observation.

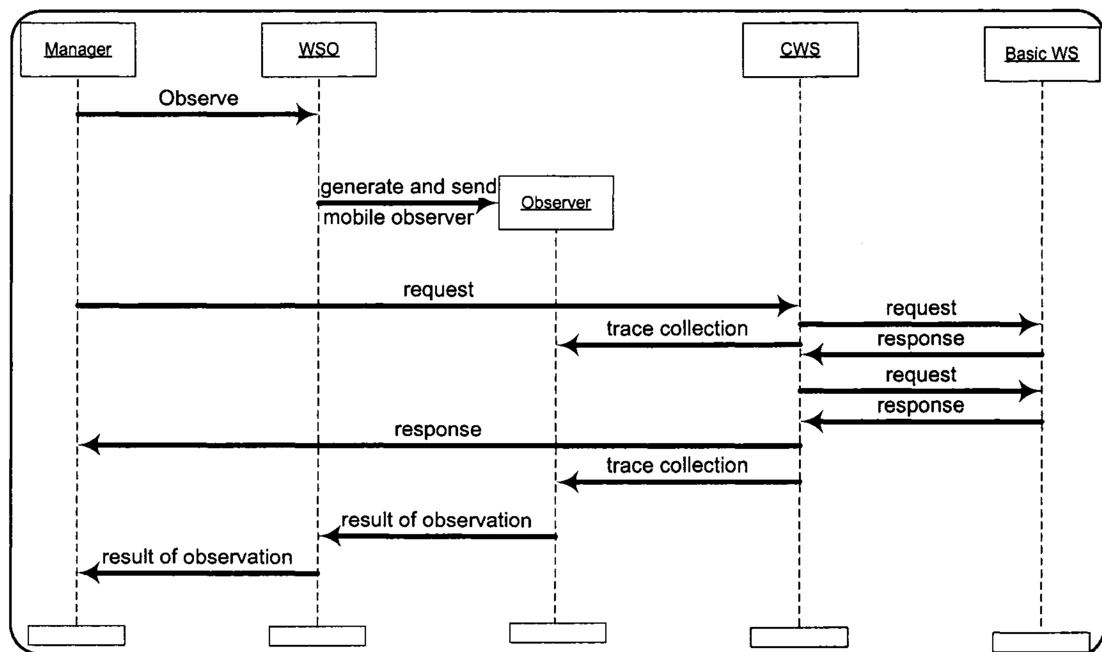

Figure 7.6 Single-observer deployment

7.4.5 Example of multi-observer observation procedure

In multi-observer architectures, in addition to the observation of the CWS, the manager needs to assure that all the steps are performed according to the agreed on contract and QoWS. Fortunately, all the providers accept to participate, to some extent, in the observation. The provider of the CWS will host all the mobile observers. It will also provide the BPEL document, WSDLs documents, and FSM+ models of each of the basic Web Services. Basic Web Services' providers will configure SOAP handlers for traces forward.

Once deployed and configured, mobile observers start their observation by performing the homing procedure. When this procedure is carried out correctly, misbehaviors detection starts. Each local observer is listening to a UDP port to receive events from SOAP handlers.

The global observer is listening to two different UDP ports: one to receive events (request or response) from local observers and another port to receive information on detected misbehaviors by local observers. Each event between a client and its Web Service is sent by the SOAP handler to the attached local observer. The latter forwards this event to the global observer and checks the validity of this event with regards to the model of the observed Web Service. If misbehavior is detected, the local observer notifies the global observer through a UDP datagram. The global observer tries to associate the new received fault with a previous fault. If the correlation fails, the global observer notifies the manager, otherwise, the misbehavior is logged and detection operations continue.

For the purpose of this case study, I extended the client application in section 6.2.2 (Figure 6.7) to allow the user to select one of the operations to invoke and provide valid or invalid parameters. Figure 7.7 shows the overall configuration of interacting client, Web Services, mobile observers, and communication between these entities.

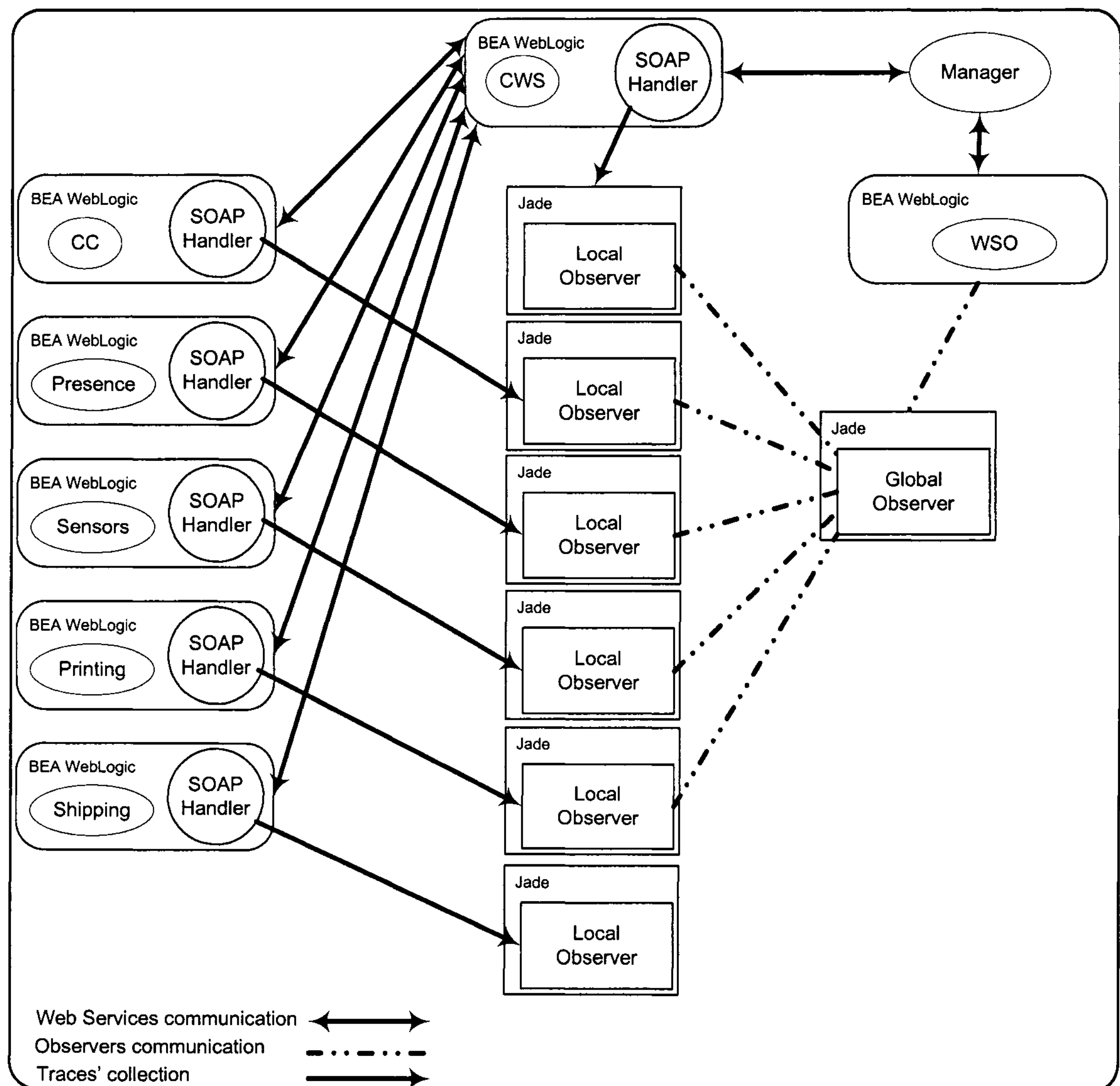

Figure 7.7 Multi-observer configuration

The observation procedure of CWS is performed following the detailed steps below and illustrated in Figure 7.8. To keep the figure simple, just one Web Service handler and one Web Service client[4] are depicted in the figure.

[4] When the composite Web Service invokes a basic Web Service, it is said to be a client of that basic Web Service.

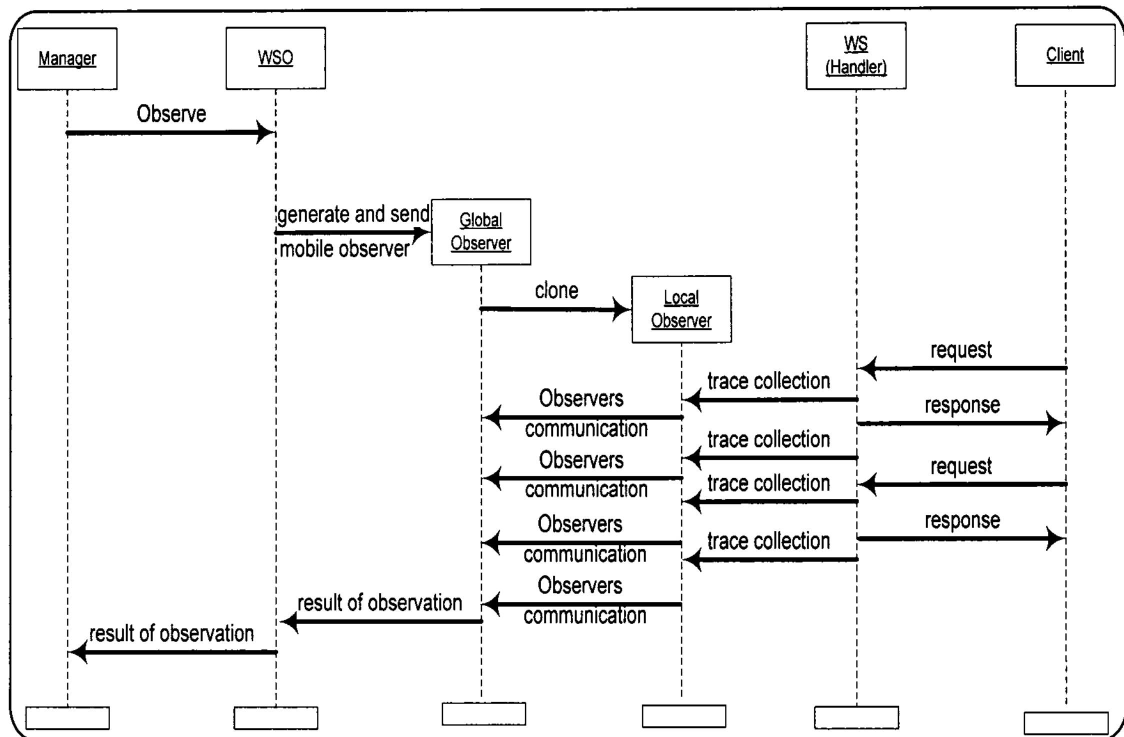

Figure 7.8 Multi-observer deployment

1. The manager invokes the WSO, providing BPEL, FSM+, and WSDL documents.
2. The WSO generates a mobile agent and sends it to appropriate destination submitted during invocation in step 1.
3. Once the mobile agent gets into its destination, it clones itself as many as required to observe all the Web Services.
4. The mobile agent with the BPEL document becomes the global observer; other mobile observers are local.
5. SOAP handlers forward traces to appropriate mobile observers.
6. Local observers observe the exchanged messages between a Web Service and its client and forward them to the global observer.

7. Whenever a misbehavior is detected (by global or local observers), correlation then fault location are initiated by the global observer to find the faulty Web Service.
8. The global observer reports to the WSO.
9. The WSO reports to the manager

7.4.6 Network load

The network load introduced by the observation is classified into two classes:

1. load due to the deployment of mobile agents and
2. load due to the trace collection process.

7.4.6.1 Deployment load

In composite Web Service's provider participation, since all observers are located at this provider's domain, only one mobile agent is generated by the Web Service Observer and sent to the hosting platform. The size of the traffic to move a mobile agent from the WSO to the composite Web Service provider is around 600 Kilobytes (600 Kb).

In basic Web Services' providers' participation, for each Web Service, a separate mobile observer will transit over the network, from the WSO to the provider of the Web Service. This represents a network load of 6 * 600 Kb (there are 6 Web Services to observe, one composite and 5 basic Web Services).

In hybrid participation, the deployment overhead depends on the number and extent of participating providers. From the configuration depicted in Figure 7.7:

- All observers are located at the provider of the composite Web Service.

- All providers deploy and configure SOAP handlers

This configuration requires moving a unique mobile observer to composite Web Service's provider. The associated overall load is again 600 Kb.

7.4.6.2 Traces' collection load

Generally, for each interaction between a Web Service and its client, 2 UDP datagrams are generated: a first datagram from the SOAP handler to a local observer, and a second datagram from this local observer to the global observer. Whenever misbehavior is detected by a local observer, a third datagram is sent (fault notification). The average size of a datagram is 150 bytes. So, each response/request pair introduces 4 datagrams if everything goes fine, 5 datagrams if one of the events is faulty, or 6 datagrams if both are faulty. We suppose that faults will not occur often, and then few fault notifications will be generated. This assumption is realistic since all Web Services are supposed to undergo an acceptable active testing process before their deployment. The trace collection load then is reduced to the forward of events, that is, 4 datagrams for a request/response pair. This represents a load of 600 bytes.

Where points of observation and mobile observers are located within the provider of the composite Web Service, traces' collection traffic is in-site[5]. If points of observation and local observers are located at basic Web Services' providers' domains, traffic from SOAP handlers to local observers is in-site and the bulk of network overhead is due to the traffic from local observers to the global observer. If points of observation are located at basic Web Services' domains and local observers are located within the provider of the composite Web Service (hybrid participation as in Figure 7.7), the overhead is associated with the traffic from SOAP

[5] As discussed above, in-site traffic is not considered in network overhead.

handlers to local observers. Messages from local observers to the global observer are in-site. Figure 7.9 shows different network overhead of different architectures when observing 2000 pairs of request/response to/from the CWS.

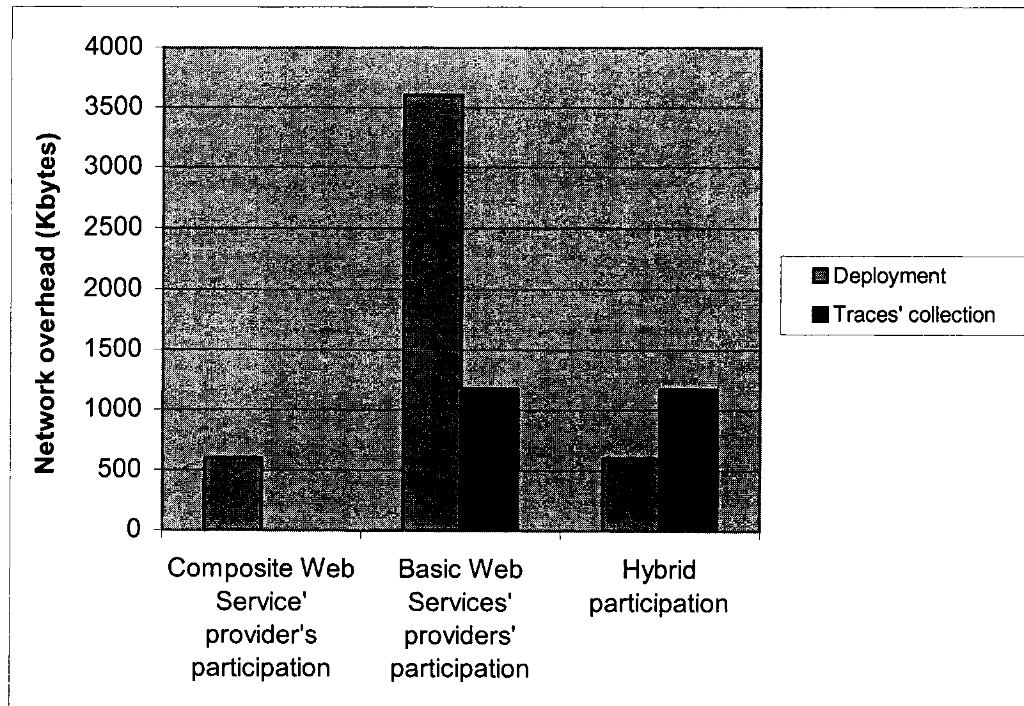

Figure 7.9 Multi-observer architectures associated network overhead

7.4.7 Results and analysis

To illustrate the detection capabilities of our architecture, we injected faults to some Web Services and or in the network and monitored the behavior of the observers. Most of the injected faults have been detected by the observers. The global observer was also able to link related notifications that are originated by the same faulty event. From the BPEL document,

the global observer builds the list of partners and the order in which they are invoked. Correlation is based on this information and the event sent within the fault notification message.

Table 7.1 shows brief descriptions of some of the executed scenarios and the reactions of observers (both local and global) to the fault.

Table 7.1 Some of the executed scenarios

Target Web Service	Fault description	Comments
CWS	Submit a printDocument request before creating a conference	Fault detected by local and global observer
Call Control	Add a user before creating a conference	Fault detected by local and global observer
Presence	Try to add a user to the conference that is not recognized by the Presence service	Fault detected by local and global observer
Shipping	Request shipping of a document that has not been submitted for printing	Fault detected by local and global observer
Shipping	A trace collection event (shipDocument response) from a handler to the local observer is lost (Figure 7.10)	Neither the local observer nor the global observer will detect the fault.
Shipping	A trace collection event (shipDocument response)	The global observer will not be able to detect the fault or process the

	or a fault notification from a local observer to the global observer is lost (Figure 7.11)	notification (correlation)

A fault that cannot be detected occurs when the last event in a communication between a Web Service and its client is lost. As discussed before, traces are sent as UDP packets. To be able to detect lost packets and recover the observation, a sequence number attribute is used. An observer detects a lost packet if the sequence number of the following received packet is different than expected. When a lost packet carries the last event in a communication, observers will not be able to detect this incident since no future packets will arrive.

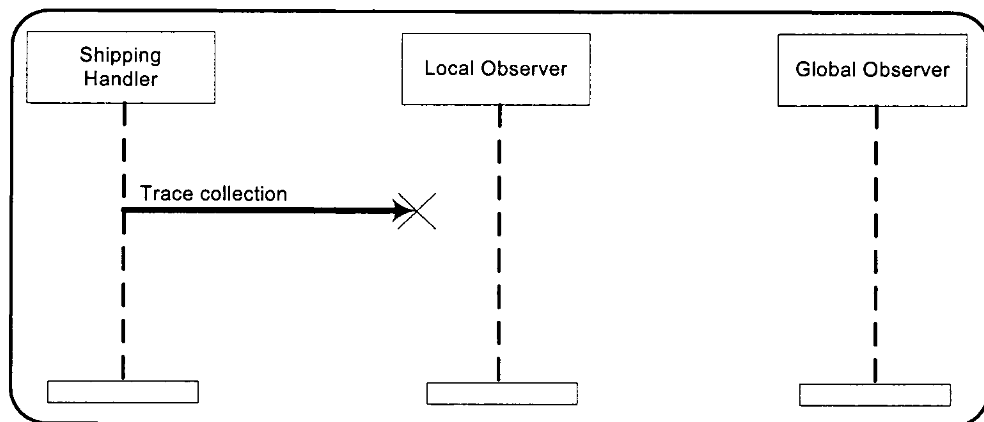

Figure 7.10 trace lost before getting to local observer

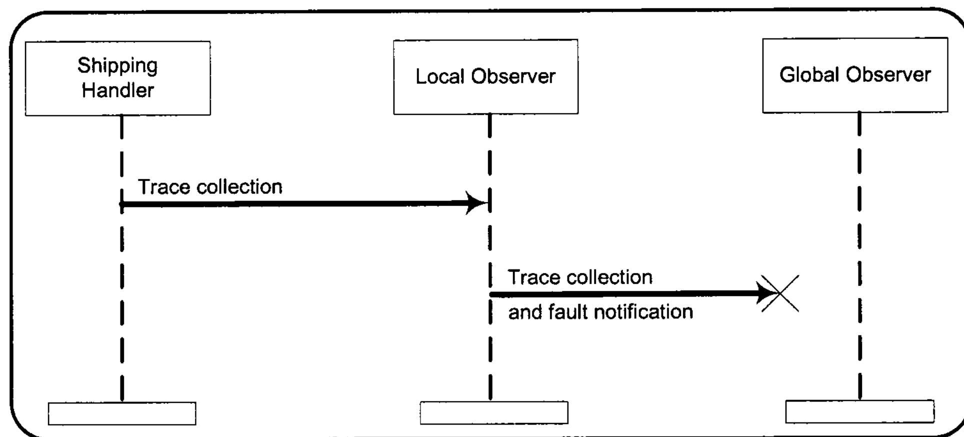

Figure 7.11 trace or fault notification lost before getting to global observer

7.5 Summary

In passive observation, the unique observation of a composite Web Service might not give insights on the behaviors of basic Web Services. Many events observed between a composite Web Service and its client cannot be studied and explained without information on the exchanged events between the composite Web Service and its basic Web Services. Thus, observation of all basic Web Services or at least a subset of these Web Services is needed.

This chapter presented multi-observer architectures for the observation of composite Web Services. The architectures propose to observe the composite Web Service and a set of basic Web Services. Heuristics to select the basic Web Services to be observed are also discussed since observation of all basic Web Services might be impractical. To reduce the network load generated by the observation, the architecture considers mobile observers. The chapter discussed also the network load in terms of mathematical complexity for each type

of participation of Web Services' providers: composite Web Service provider's participation, basic Web Services providers' participation, and hybrid participation.

As a proof of concept, I developed a composite Web Service and its basic Web Services for conferencing management. I also evaluated the network load introduced by observation and misbehaviors detection capabilities of different observers.

After dealing with control and QoWS aspects of Web Services in observation, the next chapter will tackle observation of data flows using EFSM models.

Chapter 8
EFSM-Based Observation

Originality does not consist in saying what no one else has ever said before, but in saying exactly what you think yourself -
James Stephens

As indicated in Chapter 3, few models have been used for model-based observers but most of published works on passive testing use FSM models. Although this model is appropriate for control parts of WSUO, it does not support data flow where EFSM is more appropriate for the handling of variables.

In the state of the art on EFSM-based observers, the homing procedure is either ignored or depends fully on observed messages. The first case supposes that the observation will start sharply with the interactions between the WSUO and its client. A passive observer based on this assumption will not be able to detect misbehaviors if it does not get whole traces. For the second case, the observer must wait for exchange of messages before moving forward in the homing procedure.

Since we are interested in online observation of Web Services, ignoring the homing procedure is not an option. We suppose that an EFSM-based online observer can initiate its observation at any time without having access to previously exchanged requests/responses. Such observers use actually exchanged messages for state and variables homing. This approach is efficient when the time gap between requests and responses is too short so the ob-

* Results from this chapter have been published in [104] and [105].

server will be processing traces most of its time. If this time gap is relatively high, the observer spends a significant amount of time waiting for events while valuable information can be gathered by analyzing the EFSM model of the WSUO. The example presented in section 8.3 shows a case where traces analysis alone fails to detect a fault that would have been detected if the observer was performing appropriate backward analysis of the EFSM machine of the WSUO.

This chapter presents a new approach for homing online EFSM-based observers. In this approach, the observer performs forward walks in the EFSM model whenever a new event is observed. It performs backward walks if no events show up. The backward walk is motivated by the possible valuable knowledge the observer can get by guessing what different paths could bring the WSUT to its actual position. If we suppose that the client makes sometime between requests (think time) and that the WSUT takes some time to respond (response time), the backward walk will not delay the homing.

8.1 EFSM-based observers: forward and backward walks

In client-server communication as in Web Services, it is reasonable to assume that there will be a delay between requests and responses. In one hand, the client takes time to formulate and send its request. Once a response is received, the client takes again some time to process the response and decide what to do next. At the other hand, the Web Service requires time to process a request, generate, and send its response.

To speed up the homing procedure, the observer should make a concise use of the information contained within the EFSM model in addition to the information carried by observed events. The homing algorithm can perform backward walks in the EFSM model to

guess what transitions the WSUO fired before getting into its actual state. By analyzing the set of predicates and variable definitions on these transitions, the observer can reduce the set of possible states and/or the set of possible values of variables. Performing both backward and forward walks provides a set of possible execution trees: the forward process adds execution sequences to the root of trees, and the backward process adds execution sequences to the leaf states of trees.

During the homing procedure, the observer manipulates the following entities:

- Set of Possible States *(SPS)*: this is the set of possible states with regards to what has been observed and processed up to now. At the beginning, all states are possible.
- Tree of Possible Previous States for state s *(TPPS(s))*: this tree contains possible paths that could lead to state s in the SPS. During the homing procedure, there is a TPPS for each state in the SPS.
- Set of Possible Variable Values for variable v *(SPVV(v))*: this is the set of all possible values that variable *v* can have with regards to what has been received and processed before. It consists of a list of specific values or ranges. At the beginning, all values in the definition's domain of variable *v* are possible.
- Set of Known Variables *(SKV)*: the set of known variables. A variable is said to be known if it is assigned a specific value. In this case, SPVV(v) contains one element, i.e. |SPVV(v)| == 1.
- Set of Unknown Variables *(SUV)*: the set of variables not yet known.

The next three sub-sections present in detail the processes of analyzing observed requests and responses and performing backward walks within an EFSM-based observer using backward walks.

8.1.1 Homing controller procedure

While the observer is going through the homing procedure (Algorithm 8.1), it has 3 possible options:

1. process a request that has been received by the WSUO (line 11),
2. process a response that has been sent by the WSUO (line 17), or
3. perform a one-step backward walk (line 20). In this case, the algorithm considers the event e that triggers the loop as empty.

Processing observed events has priority and the backward walk is performed if and only if there are no observed events waiting for processing. This procedure is repeated until:

- a fault is detected (unexpected input/output which results in an empty set of possible states and/or contradictory values of variables), or
- the set of possible states has one item ($|SPS| == 1$) and the set of unknown variables is empty ($SUV == \emptyset$).

```
    SPS := S // At startup, all states are possible
    SUV := V // At startup, all variables are unknown
    Expected_Event ← "Any"
    Data: event e
 4  repeat
        e ← observed event
        switch (e) do
            case (e is an input)
                if (Expected_Event == "Output") then
                    return "Fault: Output expected not Input"
                else
11                  processInput(e);          // Complexity: O(BF_PI)
                    Expected_Event ← "Output";
            case (e is an output)
                if (Expected_Event == "Input") then
                    return "Fault: Input expected not Output"
                else
17                  processOutput(e) ;        // Complexity: O(BF_PO)
                    Expected_Event ← "Input";
            otherwise
20              performBackWalk ;             // Complexity: O(BF_BW)
    until (|SPS| == 1)AND(|SUV| == 0)
```

Algorithm 8.1 Homing controller

The complexity of Algorithm 8.1 depends on the number of cycles required to successfully achieve the homing procedure (line 4) and the complexity of processInput (line 11), processOutput (line 17), and performBackWalk (line 20). Let us denote the number of cycles

to achieve the homing by n, and the complexities of processInput, processOutput and performBackWalk by O(BF_PI), O(BF_PO), and O(BF_BW) respectively. The complexity O(H) of the homing algorithm is given in Equation 1 and will be developed through the following sb-sections when individual complexities will be computed.

$$O(H) = n.O(BF_PI) + n.O(BF_PO) + n.O(BF_BW) \qquad \textbf{Equation 1}$$

8.1.2 Processing requests

When the observer witnesses an input, if the observer was expecting an output, a fault ("Output expected rather than Input") is generated. Otherwise, it removes all the states in the set of possible states that don't accept the input, and the states that accept the input but the predicate of the corresponding transition is evaluated to FALSE. For each of the remaining possible transitions, the input parameters are assigned (if applicable) to appropriate state variables. Then, the predicate condition is decomposed into elementary expressions (operands of AND/OR/XOR combinations). For each state variable, the set of possible values/ranges is updated using the elementary conditions. If this set contains a unique value, this latter is assigned to the corresponding variable; this variable is then removed from the set of unknown variables and added to the set of known variables. The transition's assignments part is processed, then updating the sets of known/unknown variables accordingly (Algorithm 8.2). The observer expects now the next observable event to be an output.

```
Input: event e
Data: boolean possibleState
1  foreach (S ∈ SPS) do
       possibleState = false
3      foreach Transition t so that ((t.S_s == S) AND (t.I == e) AND
       (t.P ≠ FALSE)) do
           possibleState = true
           assign appropriate variables the values of the parameters of e
6          update SPVV, SKV, and SUV
7          decompose the predicate into elementary conditions
8          foreach (elementary condition) do
9              update the SPVV, SKV, and SUV
               if (contradictory values/ranges) then
                   return "Contradictory values/ranges"
       if (possibleState == false) then
           remove S from SPS
           if (SPS == ∅) then
               return "Fault detected before homing is complete"
```

Algorithm 8.2 Processing observed requests

The complexity of Algorithm 8.2 is affected by the maximum number of states in the SPS (line 1), maximum number of transitions at each state in the SPS (line 3), and the complexity of updating the SPVV, SKV, and SUV. In fact, we can assume that a predicate will have very few elementary conditions, then decomposing the predicate (line 7) and using the elementary conditions to update the variables (line 8) does not affect the complexity of the

whole algorithm. If the number of variables is V, the complexity of updating the SPVV, SKV, and SUV is in the order of O(V) since the procedure should go through all the variables. The complexity of Algorithm 8.2 is depicted in Equation 2 where S_{max} is the maximum number of states in the SPS, and T_{max} is the maximum number of transitions that a state in the SPS can have.

$$O(BF_PI) = O(S_{max}.T_{max}.V) \qquad \textbf{Equation 2}$$

8.1.3 Processing responses

In case the event is a response (output), if the observer was expecting an input, a fault ("Input expected rather than Output") is generated. Otherwise, the observer removes all the states in the set of possible states that don't have transitions that produce the output. If a state has two (or more) possible transitions, the TPPS is cloned as many as possible (number of possible transitions) so that each clone represents a possible transition. The assignment part of the transition is processed and variables are updated. The set of possible states holds the ending states of all the possible transitions. In the context of SOAP communication between a Web Service and its client, the response (message) holds basically one parameter. Whenever an output message is observed, a variable becomes known, or at least a new condition on variable values is augmented unless the message carries no parameter or the variable is already known. The observer expects now the next observable event to be an input.

```
    Data: event e
    Data: boolean possibleState
 1  foreach (S ∈ SPS) do
        possibleState = false
 3      foreach Transition t so that ((t.S_s == S) AND (t.I == e) AND
        (t.P ≠ FALSE) AND (t can produce e)) do
            possibleState = true
 5          clone the corresponding TPPS
            t.S_e becomes the root of the cloned TPPS
            S becomes its child
 8          process the transition's assignment part
 9          assign appropriate variables values of the parameter (if any) of e
10          update the SPVV, SKV and SUV
            if (contradictory values/ranges) then
                return "Contradictory values/ranges"
13      remove S from the SPS
14      remove the original TPPS ;    /* no longer useful, cloned (and
        updated) trees will be used */
    if (possibleState == false) then
16      remove S from SPS
        if (SPS == ∅) then
            return "Fault detected before homing is complete"
19      remove the corresponding TPPS
```

Algorithm 8.3 Processing observed responses

Let's now determine the complexity of Algorithm 8.3. If we denote the maximum number of nodes (i.e states) in a TPPS tree by P_{max}, cloning a TPPS tree (line 5) is in the order of

$O(P_{max})$. Moreover, the complexity of removing a TPPS tree (lines 14 and 19) is also in the order of $O(P_{max})$. Lines 8 and 9 do not affect the complexity since the number of assignments in a transition is somehow low compared, for instance, to P_{max}. The complexity of Algorithm 8.3 then can be written as:

$$O(BF_PO) = O(S_{max}.T_{max}.(P_{max} + V)$$ **Equation 3**

8.1.1 Performing backward walks

While the observer is waiting for a new event (either request or response), it can perform a 1-step backward walk in the EFSM model to guess the path that could bring the Web Service to its actual state (Algorithm 8.4). From each state in the set of possible states, the observer builds a tree of probable-previously visited states and fired transitions.

Whenever a state is added to a leaf state in a TPPS, the variables constraints on the corresponding transition propagates toward the parent states (higher level states in the tree) up to the root of the tree. Two constraints on two different children of a state will propagate as operands of an *or* logic operator at their parent. While two constraints appearing in two transitions between states in successive levels of the tree (child → parent → upper parent) propagate as operands of an *and* logic operator at the upper parent.

```
Input: EFSM
1 foreach (TPPS) do
2     foreach (Leaf state S of TPPS) do
3         foreach (state S' in the EFSM that leads to S) do
4             Propagate the constraints of the corresponding transition
              toward the root of TPPS;
              /* Lets consider that propagation cannot go beyound
                 state S_p                                          */
              if (S_p is the root of TPPS) then
                  /* this path is possible→ consider it in the
                     TPPS                                           */
                  add S' as child of S;
7             update SPVV, SKV, and SUV;
              if (contradictory values/ranges) then
                  return "Contradictory values/ranges";
```

Algorithm 8.4 Performing backward walks

For example, in Figure 8.1 where the labels on transition are simplified to illustrate only the variables constraints (A, B, and C), S4 is the root of a TPPS. The first back walk step leads to state S3, the variables constraints C will propagate to S4, and state S3 becomes a leaf child of S4. During the second step of the back walk, all states preceding S3 are explored; two possible transitions: to S1 and to S2 propagating variables constraints A or B to S3. These constraints will propagate to S4 by and operator. Variables constraints at S4 after two backward steps are: ((A **or** B) **and** C).

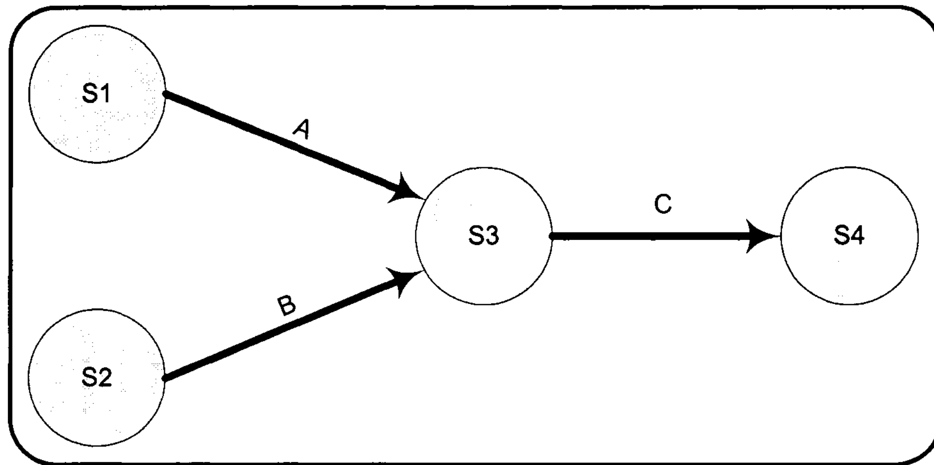

Figure 8.1 Constraints propagation

Algorithm 8.4 has three embedded loops. The first loop (line 1) is bounded by the number of TPPS trees; that is, the number of states in the SPS (S_{max}). The second loop (line 2) goes through all leaf states of a TPPS, which is, in the worst case, P_{max}. The third loop (line 3) explores all the states in the EFSM that can lead to a particular state in a TPPS. Let's denote the number of states in an EFSM by S_{EFSM}. Propagating a constraint through the root of a TPPS (line 4) is in the order of $O(P_{max}.V)$ since the procedure has to process all states and update the SPVV at each state. The complexity of Algorithm 8.4 can be written as:

$$P(BF_BW) = O(S_{\max}.S_{EFSM}.P^2{}_{\max}.V) \qquad \textbf{Equation 4}$$

From Equation 1, Equation 2, Equation 3, and Equation 4, the overall complexity for homing an observer using Algorithm 8.1 can be developed as follows:

$$O(H) = O(n.S_{\max}.T_{\max}.V + n.S_{\max}.T_{\max}.(P_{\max} + V) + n.S_{\max}.S_{EFSM}.P^2{}_{\max}.V) \quad \textbf{Equation 5}$$

$$O(H) = n.S_{\max}.T_{\max}.(P_{\max} + V) + n.S_{\max}.S_{EFSM}.P^2{}_{\max}.V)$$

The complexity of the EFSM homing algorithm with backward walks is linear as shown in Equation 5. Moreover, since backward walks are performed during absence of traces, the cost of combining backward and forward walks is minimal when compared to the detection power it adds to an EFSM-based passive observer.

8.2 Discussion

Although backward walks-based observers require a little bit more resources than an observer without backward walks, this overhead is acceptable. First of all, backward walks are performed whenever there is no trace to analyze so the observer does not use additional processing time. It just uses the slots initially allocated to trace analysis. Second, limiting the backward to a unique step at a time reduces the duration of cycles of Algorithm 8.4 and does not delay processing of eventual available traces.

As for convergence of Algorithm 8.1, it is not possible to decide if the observer will converge or not. This is the case for both brands of observers: with backward and without backward walks. This limitation is out of the scope of the homing approach used but fully tied to the fact that the observer has no control on exchanged events. The Web Service and its client can continuously exchange messages that do not bring useful information to reduce the SPS and the SUV.

However, the backward approach can be compared to the approach without backward, for the same WSUO and observed traces, as follows:

- **Theorem 1**: if an observer without backward walks converges, an observer with backward walks converges too.
- **Theorem 2**: if an observer without backward walks requires n cycles to converge, and an observer with backward walks requires m cycles to converge, then $m \leq n$.

The next paragraphs present a proof of theorem 2 which can be considered also as proof for theorem 1.

Proof

The homing algorithm converges when the SPS has one element and the SUV is empty. The SUV is empty when, for each variable v in V, SPVV(v) contains a unique element.

As discussed above, analysis of traces adds states as roots of TPPS and backward walks adds states as leaves of TPPS. Whenever a trace can generate two different execution paths, the corresponding TPPS is cloned. This will build TPPS trees where the root has a unique child. In such trees, all constraints propagation from backward walks will propagate using AND operator between the root and its child. This propagation tries to reduce the SPVV; in the worst case the SPVV is neither reduced nor extended.

In Figure 8.2, at cycle i, a TPPS has S_i as root, S_j is its child, and $SPVV_i(v)$ is the set of possible values of variable v at S_i as computed from a previously observed trace. Suppose that during cycle i+1, the backward walk adds two leaves to S_j: S_l and S_k. In Figure 8.2, the labels on transitions represent the SPVV that result from the predicate of the transitions.

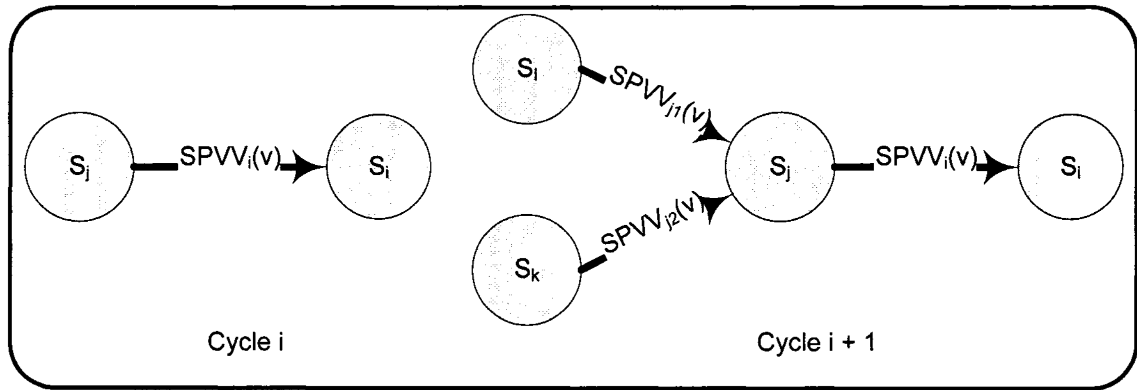

Figure 8.2 SPVV and constraints propagation

Propagation of constraints from S_l and S_k to S_j and then to S_i modifies SPVV(v) as follows: $SPVV_{i+1}(v) = SPVV_i(v) \cap ((SPVV_l(v) \cup SPVV_k(v))$. There are three cases:

1. $SPVV_i(v) \subseteq (SPVV_l(v) \cup SPVV_k(v))$: in this case, the $SPVV_{i+1}(v)$ is equal to $SPVV_i(v)$. The backward walks do not bring useful information to reduce $SPVV_i(v)$. If subsequent backward walks do the same, the number of required cycles for homing remains unchanged: *m=n*.
2. $SPVV_{i+1}(v) = \varnothing$: this indicates that variable v, at S_i after cycle i+1, cannot have any value from its definition domain. The observer detects a fault immediately without waiting for the next observed event which results in *m* strictly less than *n* ($m<n$).
3. $SPVV_{i+1}(v) \subset SPVV_i(v)$: in this case, the SPVV(v) is reduced. If following backward walks, associated to trace analysis, reduce further the SPVV(v), the homing with backward is likely to require less than *n* cycles ($m<n$) or at most *n* cycles ($m=n$).

End of proof

The following example illustrates the third case where backward walks reduce the number of required cycles (m<n) and allows detection of faults that cannot be detected without backward walks. The execution of the homing procedure is detailed hereafter in a step by step scenario.

8.3 Example

Let's consider the portion of an EFSM of a Web Service illustrated in Figure 8.3 where variables u, x, y, and z are integers. Events I1(15), O(13), and I2(0) are observed respectively. Each transition is represented as t:I|P|A|O where t is the label of the transition, I its input, P its predicate, A is the set of assignments, and O is the output. A predicate of a transition is evaluated to TRUE/FALSE if its condition is true/false; otherwise it is said INCONCLUSIVE if the predicate cannot be evaluated. The latter case occurs if some of the variables in the predicate are not yet known.

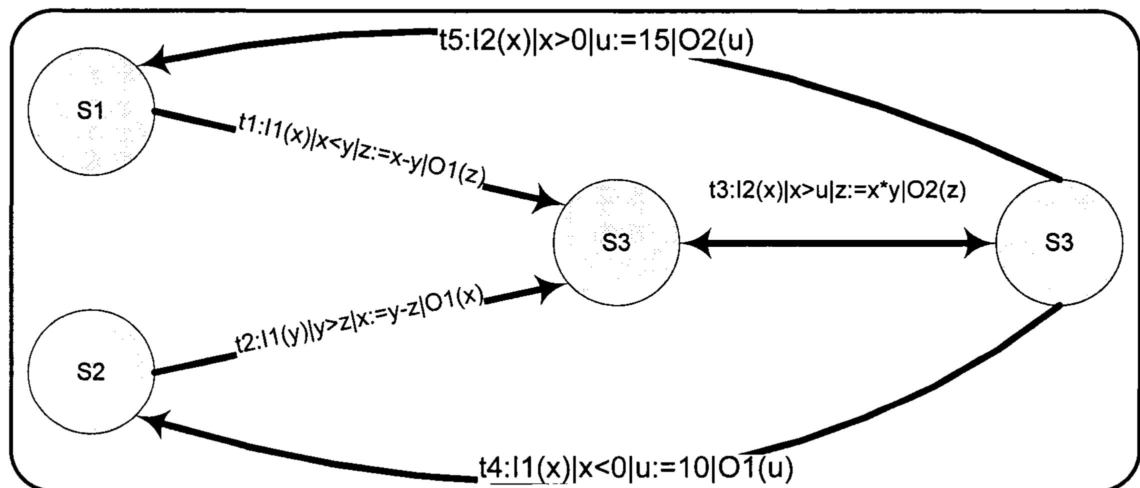

Figure 8.3 EFSM example

8.3.1 Observation without backward walks

After observing I1(15), transitions t1, t2, and t4 can be fired but not t3 or t5. However, since the input parameter is bigger than 0, the predicate of t4 is evaluated to FALSE. Only transitions t1 and t2 should be considered since the variables y and z are, up to now, unknown and the predicates are evaluated to INCONCLUSIVE. This reduces the set of possible states to S1 and S2. If t1 is executed then $x := 15$, $y > 15$, and $z := 15 - y$, if t2 is executed then $y := 15$, $z < 15$, $x := 15 - z$.

When O1(13) is observed, the value of the output parameter (13) indicates that transition t2 has been executed. Later on, when event I2(0) is observed, since the variable u is unknown, the predicate $(x > u)$ is evaluated to INCONCLUSIVE, which enables the transition. So, the sequence I1(15), O1(13), I2(0) executes properly.

However, the sequence I1(15), O1(13), I2(0) is a faulty sequence and the fault would be detected if backward walks have been considered as discussed in the next sub-section.

8.3.2 Observation with backward walks

The delay after each event (I1, O1, and I2) gives the observer opportunities to perform backward walks. The observer executes the following operations: processInput(I1(15)), performBackWalk, processOutput(O1(13)), performBackWalk, processInput(I2(0)).

As illustrated in Table 8.1, after executing the first three operations, SPS contains S3. In TPPS, S2 is the child of S3. To get to S2, the only previous transition is t4 which assigns 10 to variable u. From this point forward, the homing procedure is completed since SPS has one state and SUV is empty. Later on when receiving I2(0), transition t3 can not be fired

since its predicate (x>u) is evaluated to FALSE. The observer notifies the WSO that a fault just occurred.

Table 8.1 Content of SPS, SPVV, SKV, SUV, and TPPS

	I1(15)	Backward walk	O1(13)
SPS	S1, S2	S1, S2	S3
SPVV	t1 : x:=15, y>15, z:= x-y or t2 : y:=15, z <15, x:=y-z	t4 : u:=10 or t5 : u:=15	x:=13, y:=15, z:=2, u:=10
SKV	t1 : x, or t2 : y	u, x or u, y	x, y, z, u
SUV	t1 : y, z, u or t2 : x, z, u	y, z or x, z	∅
TPPS	Figure 8.4.a	Figure 8.4.b	

TPPS trees after each operation are illustrated in Figure 8.4.

8.4 Summary

The homing procedure in EFSM-based observers consists of recognizing the actual state of the WSUO in addition to assigning appropriate values to different variables. Common approaches either ignore this phase of passive testing or base it on upcoming observed request/responses. In online observers, the first option represents a serious limitation and the second option is not always efficient.

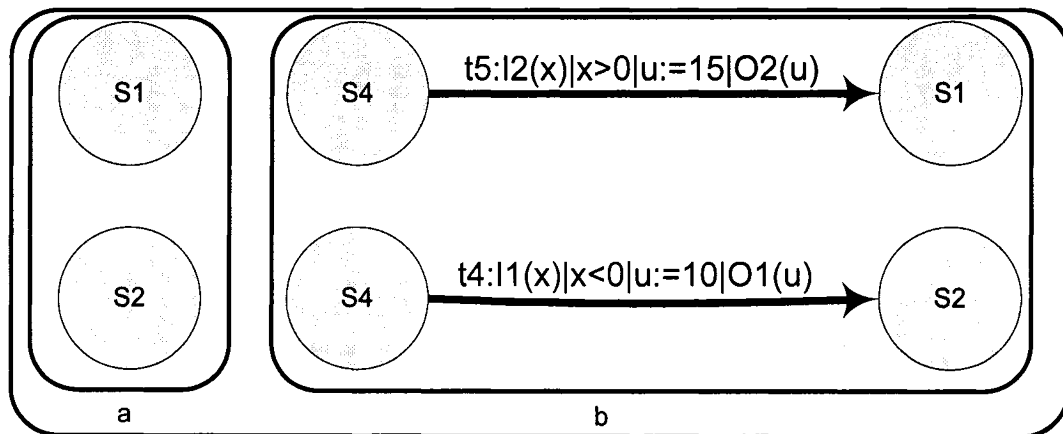

Figure 8.4 TPPS

This chapter presented a novel approach for homing EFSM-based observers. This approach is based on observed events and on backward walks in the EFSM model of the WSUO. Whenever a trace is observed, it's immediately processed by the observer. Otherwise, the observer analyzes the possible paths that could bring the WSUO to its actual state. Analyzing the set of constraints on different paths could reduce the set of possible values variables can have at a specific state.

Chapter 9
Conclusion and Future Work

There will come a time when you believe everything is finished. That will be the beginning -
Louis L'Amour

9.1 Summary of thesis

The emergence of Web Services concepts changed the way businesses interact over the Internet. The low level of coupling in Web Services allows easier and straightforward interoperability between Web Services and their clients. Composition of Web Services offers a new trend for code reusability by using available Web Services to provide new Web Services with richer functionalities.

The wide spread utilization of Web Services implied an urgent need for their management. Web Services providers and their clients appealed for management architectures from the earlier days of Web Services paradigm. However, actors in the industry were more interested in specification languages and development and deployment platforms. Later on, when they started developing management approaches, they designed them to be integrated into hosting platforms. There are three major limitations to such approaches:

1. Since they are integral parts of hosting architectures, their utilization is reserved to Web Services' providers and cannot be used by clients, and/or
2. They are based on active testers; so they cannot be used online to assess the correctness of interactions between a Web Service and a client, and/or

3. Using such approaches requires buying appropriates tools/licenses, installing, and configuring them on dedicated hosts; hence, increasing providers' costs and investments.

In this thesis, we are interested in management of synchronous Web Services. We first started by investigating the state of the art of management of Web Services. Our study showed that there is a need for management architecture that allows on-the-fly transparent detection of misbehaviors, low cost, platform-independent, and open to providers, clients, and third party certification entities. We proposed and experimented then novel management architectures that solve some of these limitations as summarized in the next section.

9.2 Thesis contributions

In this thesis, I presented an incremental series of contributions to the management of Web Services as pointed out by Figure 9.1. Since early beginning of this thesis, the concern of successful management of Web Services as a key condition for their success has been made clear and of prime importance. From that point forward, we studied available existing approaches for management of Web Services and identified their limitations.

The first contribution proposes to extend the Service Oriented Architecture (SOA) with observation capabilities by developing the observer as a Web Service. First, this will simplify interactions between the observer and entities interested in observation. The interactions use standardized technologies, and interoperability problems are unlikely to happen. Moreover, since the observer is a Web Service, the observation system is not tied to the calling entity or the platform hosting the Web Service to be observed. Second, since monitoring is more or less a temporary activity, there is no need to develop its own tester: the entity interested in

testing can invoke the Web Service observer whenever required instead of developing it, or buying a full license of a commercial testing system that will be used intermittently.

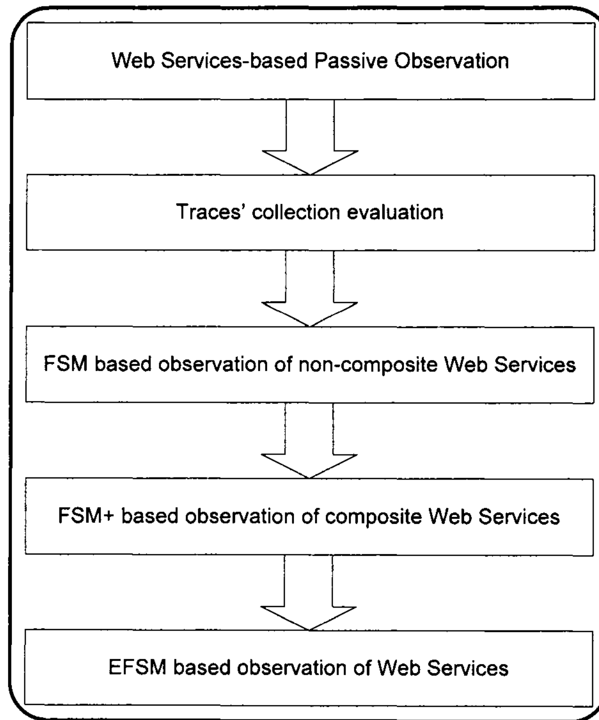

Figure 9.1 Chronology of thesis contributions

The second contribution is an observation architecture that allows observation of basic (non composite) Web Services and evaluation of the effectiveness and overhead of the proposed architecture. A thorough study of potential traces' collections mechanisms is conducted. This study, where observers are modeled by Finite State Machine (FSM), showed that mobile agents present the lowest overhead. A case study where different valid and invalid scenarios have been executed showed promising results in terms of detection capacities of the architecture.

Another contribution allows observation of composite Web Services. When it comes to observation of a composite Web Service, a single observer cannot appropriately handle observation of different Web Services. Several architectures are proposed to observe a composite Web Service. The main differences between these architectures are related to the location of observers, location of points of observation, and participation of involved Web Services' providers and clients. We extended the FSM model by a set of QoWS properties of Web Services so the observer will be able to detect performance violations as well as functional correctness. We designated the new model by FSM+. The overhead of each of the architectures is analytically discussed and then illustrated through a case study where a composite Web Service is observed.

Even if FSM and FSM+ models can represent appropriately the control part and performance properties of a Web Service, they do not support data flows. Since data flows are an important aspect in XML–based communication as in Web Services, we propose EFSM-based observation. However, the homing of an EFSM-based observer is far harder than FSM-based observer. In fact, state variables in the EFSM model have to be correctly initialized before any detection can take place. The approaches in the state of the art are not always efficient especially if the time gap between requests and responses is somehow significant. I proposed a new homing approach to make use of this time gap to speed up the homing by performing backward walks in the EFSM models. With this approach, the detection of faults is enhanced.

An observer should have enough knowledge on the expected behavior of a Web Service and a client in order to detect misbehaviors. Behaviors' descriptions should cover both func-

tional and non-functional (QoWS) aspects. For the first element, we studied different models including FSM and EFSM. For QoWS, potential options comprise specifying QoWS attributes in dedicated documents, adding them to WSDL, FSM/EFSM, or to the BPEL of a composite Web Service.

9.3 Future work

There are still improvements and extensions to be done in the field of management of Web Services. In the upcoming future work, I am planning to tackle the following research issues:

- Enhance the detection capabilities of the observers using hybrid observers. Instead of using a model-based observer, I am planning to develop a new brand of passive observers based on combination of models and knowledge bases/inference engines.
- Fault isolation is very vital to business-to-business communication. In fact, a faulty Web Service should be isolated as soon as possible to prevent affecting other Web Services. The propagation of a fault can damage all interacting entities, resulting in big business loss. The architecture proposed above already locates a faulty Web Service, and I am looking at developing techniques to isolate a faulty Web Service without disturbing overall transactions
- Once a faulty Web Service is isolated, it should be replaced if possible by an equivalent/duplicate Web Service. This is mainly useful for composite Web Services. I will extend the initial Web Service architecture by adding operations to switch from a faulty Web Service to a replacement Web Service.

- The isolated Web Service should be repaired. I am planning to integrate the information provided by management architectures into the new Web Services Development Life Cycle so the faulty module (within the faulty Web Service) can be located and corrected accurately.
- During the last two years, I concentrated on the utilization of mobile agents as a mechanism of traces collection when observing a composite Web Service. I will consider other traces collection mechanisms to suit different clients' needs. For example, a client without a mobile agent platform, but with instrumented Web Services or SNMP agents, can also make use of the management architecture.
- The information gathered from management of Web Services (both functional and non-functional) must be used to advice clients to select the appropriate Web Service from an available list. This will bring competition between Web Services providers and benefits the clients. For example, a client can decide which criteria are important during the find operation (e.g. good reputation, previous failures, QoWS, low cost...). I will propose a representation model of this information and how it can be compiled and latter on used by clients.
- A research issue that I intend to study in the future is related to the consideration of security when observing Web Services. A communication between a client and a Web Service should not be spied without the consent of at least one of them.

Bibliography

[1] IBM, "WebSphere", at http://www-306.ibm.com/software/websphere/, Visited on 2007.

[2] BEA, "WebLogic plateform", at http://www.bea.com, Visited on 2004.

[3] Microsoft, ".Net", at http://www.microsoft.com/net/, Visited on 2007.

[4] Parasoft, "SOATest", at http://www.parasoft.com/jsp/products/home.jsp?product=SOAP, Visited on 2006.

[5] Parasoft, ".Test", at http://www.parasoft.com/jsp/products/article.jsp?label=product_info_TestNet, Visited on 2007.

[6] Parasoft, "WebKing", at http://www.parasoft.com/jsp/products/home.jsp?product=WebKing&, Visited on 2007.

[7] U. Wahli, "Self-study guide: WebSphere Studio Application Developer and web services", IBM Corp., International Technical Support Organization, San Jose, Calif., Text. at http://www.books24x7.com/marc.asp?isbn=0738424196

[8] WSDL, "Web Services Description Language", at http://www.w3c.org/TR/wsdl, Visited on 2003.

[9] XML, "eXtensible Markup Language", at http://www.w3c.org/XML, Visited on 2006.

[10] 3GPP, "Open Service Architecture WorkGroup", at http://www.3gpp.org/TB/CN/CN5/CN5.htm, Visited on 2003.

[11] CPXe, "I3A Standards - Initiatives - CPXe", at http://www.i3a.org/i_cpxe.html, Visited on 2003.

[12] SOA, "Service Oriented Architecture", at http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/, Visited on 2004.

[13] H. Kreger, "Web Services Conceptual Architectures (WSCA 1.0)", While Paper, *IBM Software Group*, 2001.

[14] OASIS, "Universal Description, Discovery, and Integration", at http://www.uddi.org/, Visited on 2007.

[15] F. Leymann, "Web service flow language (WSFL) 1.0", at http://www-4.ibm.com/software/solutions/webservices/pdf/WSFL.pdf, Visited on 2007.

[16] A. Ankolekar, M. Burstein, J. R. Hobbs, O. Lassila, D. Martin, D. McDermott, and S. A. McIlraith, "DAML-S: Web Service description for the Semantic Web", *First International Web Conference, Lecture Notes in Computer Science*, pub: Springer Verlag. Sardinia, Italy, 2002, pp. 348-63.

[17] A. Banerji, C. Bartolini, D. Beringer, V. Chopella, K. Govindarajan, A. Karp, H. Kuno, M. Lemon, G. Pogossiants, S. Sharma, and S. Williams, "WSCL: The Web Services conversation language", at http://www.w3.org/TR/wscl10/, Visited on 2007.

[18] A. Arkin, S. Askary, S. Fordin, W. Jekeli, K. Kawaguchi, D. Orchard, S. Pogliani, K. Riemer, S. Struble, P. Takacsi-Nagy, I. Trickovic, and S. Zimek, "Web Service Choreography Interface (WSCI)", at http://www.w3.org/TR/wsci/, Visited on 2007.

[19] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana, "BPEL4WS Version 1.1 specification", at ftp://www6.software.ibm.com/software/developer/library/ws-bpel.pdf, Visited on 2007.

[20] SOAP, "Simple Object Access Protocol", at http://www.w3c.org/TR/soap, Visited on 2004.

[21] ISO/IEC, "7498, Information processing systems -- Open Systems Interconnection - Basic Reference Model", 1989.

[22] IETF, "Internet Protocol", at www.ietf.org, Visited on 2007.

[23] ISO/IEC, "7498-4 Information processing systems -- Open Systems Interconnection - Basic Reference Model -- Part 4: Management framework", 1989.

[24] R. Dssouli, K. Saleh, E. Aboulhamid, A. En-Nouaary, and C. Bourhfir, "Test development for communication protocols: towards automation," *Computer Networks*, vol. 31, pp. 1835-72, 1999.

[25] S. Naito and M. Tsunoyama, "Fault detection for sequential machines by Transition-Tours", *11th Annual International Symposium on Fault-Tolerant Computing*, pub: IEEE. Portland, ME, USA, 1981, pp. 238-43.

[26] G. Gonenc, "A method for the design of fault detection experiments," *IEEE Transactions on Computers*, vol. C-19, pp. 551-8, 1970.

[27] T. S. Chow, "Testing software design modeled by finite-state machines," *IEEE Transactions on Software Engineering*, vol. SE-4, pp. 178-87, 1978.

[28] A. Benharref, Z. Berbich, R. Dssouli, and I. Chrisment, "Formal Specification, TTCN and executable test cases for main IPv6 protocols", *IEEE 2nd International Symposium on Signal Processing and Information Technology*. Marrakech, Morocco, 2002.

[29] J. M. Wing, "A specifier's introduction to formal methods," *IEEE Computer*, vol. 23, pp. 8-10, 1990.

[30] K. J. Turner, *Using formal description techniques : an introduction to Estelle, LOTOS, and SDL*. Chichester; New York: Wiley, 1993.

[31] ISO/IEC, "9646, Conformance Testing Methodology and Framework", 1996.

[32] M. Diaz, G. Juanole, and J. P. Courtiat, "Observer-a concept for formal on-line validation of distributed systems," *IEEE Transactions on Software Engineering*, vol. 20, pp. 900-13, 1994.

[33] K. Vijayananda and P. Raja, "Models of Communication Protocols for Fault Diagnosis", *Swiss Federal Institute of Technology* November 1994.

[34] D. Brand and P. Zafiropulo, "On Communicating Finite-State Machines," *J. ACM*, vol. 30, pp. 323-342, 1983.

[35] W3C, "Web Services Management Concern", *W3C Consortium*, White Paper November 2002.

[36] J. Case, M. Fedor, M. Schoffstall, and J. Davin, "RFC 1157 - Simple Network Management Protocol (SNMP)", IETF, Ed., 1990.

[37] D. Perkins and E. McGinnis, *Understanding SNMP MIBs*. Upper Saddle River, N.J.: Prentice Hall PTR, 1997.

[38] ISO/IEC, "9596, Information technology -- Open Systems Interconnection -- Common management information protocol", 1998.

[39] U. Warrier, L. Besaw, L. LaBarre, and B. Handspicker, "RFC 1189 - The Common Management Information Services and Protocols for the Internet (CMOT and CMIP)", IETF, Ed., 1990.

[40] W3, "Web Services Endpoint Management Architecture Requirements", at http://dev.w3.org/cvsweb/2002/ws/arch/management/ws-arch-management-requirements.html?rev=1.7.

[41] J. A. Farrell and H. Kreger, "Web services management approaches," *IBM Systems Journal*, vol. 41, pp. 212-27, 2002.

[42] F. Casati, E. Shan, U. Dayal, and M.-C. Shan, "Business - Oriented management of Web Services," *Communications of the ACM*, vol. 46, pp. 55-60, 2003.

[43] HP, "Open View", at http://www.managementsoftware.hp.com, Visited on 2007.

[44] B. Xiaoying, D. Wenli, T. Wei-Tek, and C. Yinong, "WSDL-based automatic test case generation for Web services testing", *International Workshop on Service-Oriented System Engineering*, pub: IEEE Computer Society. Beijing, China, 2005, pp. 207-12.

[45] Y. Jiang, G.-M. Xin, J.-H. Shan, L. Zhang, B. Xie, and F.-Q. Yang, "A method of automated test data generation for Web services," *Chinese Journal of Computers*, vol. 28, pp. 568-77, 2005.

[46] R. Siblini and N. Mansour, "Testing web services", *International Conference on Computer Systems and Applications*, pub: IEEE. Cairo, Egypt, 2005, pp. 763-770.

[47] K. ChangSup, K. Sungwon, K. In-Young, B. Jongmoon, and C. Young-Il, "Generating test cases for Web services using extended finite state machine", *IFIP International Conference on Testing of Communicating Systems (TestCom), Lecture Notes in Computer Science*, pub: Springer Verlag. New York, NY, USA, 2006, pp. 103-17.

[48] A. Tarhini, H. Fouchal, and N. Mansour, "A simple approach for testing Web service based applications", *Innovative Internet Community Systems. 5th International Workshop, IICS 2005. Revised Papers (Lecture Notes in Computer Science Vol.3908)*, pub: Springer-Verlag. Paris, France, 2006, pp. 134-46.

[49] J. Garcia-Fanjul, C. de la Riva, and J. Tuya, "Generation of conformance test suites for compositions of Web services using model checking", *Proceedings. Testing: Academic and Industrial Conference - Practice and Research Techniques*, pub: IEEE Computer Society. Windsor, UK, 2006, pp. 4 pp.

[50] W3C, "QoS for Web Services: Requirements and Possible Approaches", at.

[51] A. ShaikhAli, O. F. Rana, R. Al-Ali, and D. W. Walker, "UDDIe: an extended registry for Web services", *Symposium on Applications and the Internet Workshops (SAINT)*, pub: IEEE Computer Society. Orlando, FL, USA, 2003, pp. 85-9.

[52] N. E. Fenton and S. L. Pfleeger, *Software metrics : a rigorous and practical approach*, 2nd ed. Boston: PWS Pub., 1997.

[53] A. R. Gray and S. G. MacDonell, "Comparison of techniques for developing predictive models of software metrics," *Information and Software Technology*, vol. 39, pp. 425-437, 1997.

[54] W. J. Salamon and D. R. Wallace, "Quality Characteristics and Metrics for reusable Software", *National Institute of Standards and Technology*, May 1994.

[55] B. Hailpern and P. L. Tarr, "Software Engineering for Web Services: A Focus on Separation of Concerns", *IBM Research Report*, September 2001.

[56] A. Mani and A. Nagarajan, "Understanding quality of service for Web services", at http://www-106.ibm.com/developerworks/library/ws-quality.html, Visited on 2002.

[57] S. Ran, "A Framework for Discovering Web Services with Desired Quality of Services Attributes", *International Conference on Web Services*, pub: CSREA Press. Las Vegas, Nevada, United States, 2003, pp. 208-213.

[58] M. A. Serhani, R. Dssouli, A. Hafid, and H. Sahraoui, "A QoS broker based architecture for efficient web services selection", *International Conference on Web Services(ICWS)*,

vol. 2005, pub: IEEE Computer Society. Orlando, FL, United States, 2005, pp. 113-120.

[59] S. Kalepu, S. Krishnaswamy, and S. W. Loke, "Verity: a QoS metric for selecting Web services and providers", *Fourth International Conference on Web Information Systems Engineering Workshops*, pub: IEEE Computer Society. Rome, Italy, 2004, pp. 131-9.

[60] W. T. Tsai, R. Paul, Z. Cao, L. Yu, and A. Saimi, "Verification of Web services using an enhanced UDDI server", *Eighth IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, pub: IEEE. Guadalajara, Mexico, 2003, pp. 131-8.

[61] S. Ho, W. M. Loucks, and A. Singh, "Monitoring the performance of a Web service", *IEEE Canadian Conference on Electrical and Computer Engineering*, vol. 1, pub: IEEE press. Waterloo, Ontario, Canada, 1998, pp. 109-12.

[62] J. Yuming, T. Chen-Khong, and K. Chi-Chung, "Challenges and approaches in providing QoS monitoring," *International Journal of Network Management*, vol. 10, pp. 323-34, 2000.

[63] A. Schmietendorf, R. Dumke, and D. Reitz, "SLA management - Challenges in the context of Web-service-based infrastructures", *IEEE International Conference on Web Services (ICWS)*, pub: IEEE Computer Society. San Diego, CA, United States, 2004, pp. 606-613.

[64] D. Lee, A. N. Netravali, K. K. Sabnani, B. Sugla, and A. John, "Passive testing and applications to network management", *International Conference on Network Protocols*, pub: IEEE Computer Society. Atlanta, GA, USA, 1997, pp. 113-22.

[65] J. A. Arnedo, A. Cavalli, and M. Nunez, "Fast testing of critical properties through passive testing", *15th IFIP International Conference on Testing of Communicating Systems, Lecture Notes in Computer Science (LNCS)*, pub: Springer Verlag. Sophia Antipolis, France, 2003, pp. 295-310.

[66] R. E. Miller, "Passive testing of networks using a CFSM specification", *International Performance, Computing and Communications Conference*, pub: IEEE. Tempe/Phoenix, AZ, USA, 1998, pp. 111-16.

[67] R. E. Miller and K. A. Arisha, "On fault location in networks by passive testing", *International Performance, Computing, and Communications Conference*, pub: IEEE. Phoenix, AZ, USA, 2000, pp. 281-7.

[68] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization of test information to assist fault localization", *International Conference on Software Engineering (ICSE)*, pub: IEEE Computer Society. Orlando, FL, United States, 2002, pp. 467-477.

[69] R. E. Miller and K. A. Arisha, "Fault identification in networks by passive testing", *34th Annual Simulation Symposium*, pub: IEEE Computer Society. Seattle, WA, USA, 2001, pp. 277-84.

[70] B. Sarikaya, "Protocol test generation, trace analysis and verification techniques", *2nd Workshop on Software Testing, Verification, and Analysis*, pub: IEEE Computer Society. Banff, Alberta., Canada, 1988, pp. 123-3.

[71] G. V. Bochmann, R. Dssouli, and J. R. Zhao, "Trace analysis for conformance and arbitration testing," *IEEE Transactions on Software Engineering*, vol. 15, pp. 1347-56, 1989.

[72] R. Borgeest and C. Rodel, "Trace analysis with a relational database system", *4th Euromicro Workshop on Parallel and Distributed Processing*, pub: IEEE Computer Society. Braga, Portugal, 1996, pp. 243-50.

[73] K. C. Tai, "Race analysis of traces of asynchronous message-passing programs", *International Conference on Distributed Computing Systems*, pub: IEEE. Baltimore, MD, USA, 1997, pp. 261-268.

[74] A. b. N. Ghedamsi, *Diagnostic tests for protocol implementations modeled by finite state machines*. Montréal: Ph.D Thesis, Université de Montréal, 1992.

[75] S. Boumaraf, *Diagnostic des protocoles de communication fondé sur les automates à états finis étendus*: Master Thesis, Université de Montréal, 2000.

[76] Y. Penncolé, *Diagnostic Décentralisé de Systèmes à événement discrets : Application aux Réseaux de Télécommunications*: Ph.D Thesis, University of Rennes 1, 2002.

[77] L. Rozé and M.-O. Cordier, "Diagnosing Discrete-Event Systems: Extending the "Diagnoser Approach" to Deal with Telecommunication Networks," *Discrete Event Dynamic Systems*, vol. 12, pp. 43-81, 2002.

[78] A. Cavalli, C. Gervy, and S. Prokopenko, "New Approaches for Passive Testing using an Extended Finite State Machine Specification", *Concordia Prestigious Workshop on Communication Software Engineering*. Montreal, 2001, pp. 225-250.

[79] A. Cavalli, C. Gervy, and S. Prokopenko, "New approaches for passive testing using an Extended Finite State Machine specification," *Information and Software Technology*, vol. 45, pp. 837-852, 2003.

[80] D. Lee, C. Dongluo, H. Ruibing, R. E. Miller, W. Jianping, and Y. Xia, "A formal approach for passive testing of protocol data portions", *10th International Conference on Network Protocols*, pub: IEEE Computer Society. Paris, France, 2002, pp. 122-31.

[81] D. Lee, C. Dongluo, H. Ruibing, R. E. Miller, W. Jianping, and Y. Xia, "Network protocol system monitoring-a formal approach with passive testing," *IEEE/ACM Transactions on Networking*, vol. 14, pp. 424-37, 2006.

[82] B. Alcalde, A. Cavalli, D. Chen, D. Khuu, and D. Lee, "Network protocol system passive testing for fault management: a backward checking approach", *Formal Techniques for Networked and Distributed Systems (FM), Lecture Notes in Computer Science (LNCS)*, pub: Springer Verlag. Madrid, Spain, 2004, pp. 150-66.

[83] B. T. Ladani, B. Alcalde, and A. Cavalli, "Passive testing - a constrained invariant checking approach", *17th International Conference on Testing of communicating systems*

(TestCom), *Lecture Notes in Computer Science (LNCS)*, pub: Springer Verlag. Montreal, Que., Canada, 2005, pp. 9-22.

[84] A. Benharref, R. Glitho, and R. Dssouli, "A Web Service Based-Architecture for Detecting Faults in Web Services", *Integrated Management*, pub: IEEE Press. Nice, France, 2005.

[85] A. Benharref, R. Glitho, and R. Dssouli, "Mobile agents for testing Web services in Next Generation Networks", *MATA 2005, Lecture Notes in Computer Science*, vol. 3744, pub: Springer Verlag. Montreal, Canada, 2005, pp. 182-191.

[86] A. Benharref, M. A. Serhani, R. Dssouli, and R. Glitho, "Les Agents Mobiles pour la Vérification de la QdS des Services Web", *6ème Conférence Internationale sur les NOuvelles TEchnologies de la REpartition (NOTERE)*, pub: Lavoisier. Toulouse, France, 2006, pp. 339-350.

[87] M. A. Serhani, R. Dssouli, A. Benharref, H. Sahraoui, and M. E. Badidi, "Toward integration of Quality of Web services (QoWS) Management operations in SOA: Case of basic and composite Web services", *selected as Book Chapter in Advanced Topics in Intelligent Information Technologies*. 2007.

[88] M. A. Serhani, R. Dssouli, H. Sahraoui, A. Benharref, and M. E. Badidi, "VAQoS: architecture for end-to-end QoS management of value added Web services," *International Journal of Intelligent Information Technologies*, vol. 2, pp. 37-56, 2006.

[89] M. A. Serhani, A. Benharref, R. Dssouli, and H. Sahraoui, "CompQoS: Towards an Architecture for QoS composition and monitoring (validation) of composite web services", *International Conference on Web Technologies, Application, and Services "WTAS"*. Calgary, Canada, 2006, pp. 78-83.

[90] M. A. Serhani, R. Dssouli, H. Sahraoui, A. Hafid, and A. Benharref, "Toward a new web services development life cycle", *International Multi-Conferences in Computer Science & Computer Engineering, International Symposium on Web Services and Applications*. Las Vegas, Nevada, USA, 2005, pp. 94-103.

[91] M. A. Serhani, R. Dssouli, H. Sahraoui, A. Benharref, and M. E. Badidi, "QoS Integration in Value Added Web Services", *2nd international conference on Innovations in Information Technology*. Dubai, U.A.E, 2005.

[92] S. Some, R. Dssouli, and J. Vaucher, "From scenarios to timed automata: building specifications from users requirements", *Asia Pacific Software Engineering Conference*, pub: IEEE Comput. Soc. Brisbane, Qld., Australia, 1995, pp. 48-57.

[93] P. Hsia, J. Samuel, J. Gao, D. Kung, Y. Toyoshima, and C. Chen, "Formal approach to scenario analysis," *IEEE Software*, vol. 11, pp. 33-41, 1994.

[94] Y. He, D. Amyot, and A. W. Williams, "Synthesizing SDL from Use Case Maps:An Experiment", *SDL 2003: System Design, Lecture Notes in Computer Science*, vol. 2708, pub: Springer Verlag. 2003, pp. 117-136.

[95] D. Harel and H. Kugler, "Synthesizing state-based object systems from LSC specifications," *International Journal of Foundations of Computer Science*, vol. 13, pp. 5-51, 2002.

[96] R. Mizouni, A. Salah, S. Kolahi, and R. Dssouli, "Composition of use cases using synchronization and model checking", *Lecture Notes in Computer Science*, vol. 4229, pub: Springer Verlag. Paris, France, 2006, pp. 292-306.

[97] A. Salah, R. Mizouni, R. Dssouli, and B. Parreaux, "Formal composition of distributed scenarios", *Formal Techniques for Networked and Distributed Systems (FORTE) Lecture Notes in Computer Science*, vol. 3235, pub: Springer-Verlag. Madrid, Spain, 2004, pp. 213-28.

[98] A. Fuggetta, G. P. Picco, and G. Vigna, "Understanding code mobility," *IEEE Transactions on Software Engineering*, vol. 24, pp. 342-61, 1998.

[99] Joana Da Silva, Khalid Hassan, Roch Glitho, and F. Khendek, "WEB Services for Conferencing in 3G Networks: A Parlay Based Implementation", *Proceedings of the International Conference on service delivery in Networks (ICIN'04)*. Bordeaux, France, 2004, pp. 245-250.

[100] Parlay, "Parlay 4.1 Specification", at www.parlay.org, Visited on 2004.

[101] Jade, "Java Agent DEvelopment Framework", at http://jade.tilab.com, Visited on 2007.

[102] A. Benharref, R. Dssouli, R. Glitho, and M. A. Serhani, "Towards the testing of composed web services in 3rd generation networks", *IFIP International Conference on Testing of Communicating Systems (TestCom), Lecture Notes in Computer Science*, vol. 3964, pub: Springer Verlag. New York, NY, United States, 2006, pp. 118-133.

[103] A. Benharref, M. A. Serhani, R. Glitho, and R. Dssouli, "Une architecture Multi-Observateur pour l'Observation des Services Web Composés", *5ème Conférence Internationale sur les NOuvelles TEchnologies de la REpartition (NOTERE)*. Gatineau, Quebec, Canada, 2005, pp. 153-162.

[104] A. Benharref, M. A. Serhani, R. Dssouli, A. En-Nouaary, and R. Glitho, "Traversée Parallèle pour l'Accélération du Test Passif des Services Web Basé sur les MEFE", *7ème Conférence Internationale sur les NOuvelles TEchnologies de la REpartition (NOTERE)*, pub: Hermès. Marrakech, Maroc, 2007, pp. 329-340.

[105] A. Benharref, R. Dssouli, M. A. Serhani, A. En-Nouaary, and R. Glitho, "New approach for EFSM-based Passive Testing of Web Services", *19th IFIP International Conference on Testing of Communicating Systems (TestCom), Lecture Notes in Computer Science*, pub: Springer Verlag. Tallin, Estonia, 2007, pp. 13-27.

Publications Related to This Thesis

[1] A. Benharref, R. Dssouli, M.A. Serhani, et R. Glitho, "Passive Testing of Web Services". *Submitted to Information and Software Technology*, Elsevier.

[2] A. Benharref, M. A. Serhani, M. Salem, et R. Dssouli, "Multi-tier Framework for Management of Web Services' Quality". Proposal accepted as book chapter in *Managing Web Service Quality: Measuring Outcomes and Effectiveness.*

[3] A. Benharref, R. Dssouli, M. A. Serhani, A. En-Nouaary, and R. Glitho, "New approach for EFSM-based Passive Testing of Web Services", *19th IFIP International Conference on Testing of Communicating Systems (TestCom), Lecture Notes in Computer Science*, pub: Springer Verlag. Tallin, Estonia, 2007, pp. 13-27.

[4] A. Benharref, M. A. Serhani, R. Dssouli, A. En-Nouaary, and R. Glitho, "Traversée Parallèle pour l'Accélération du Test Passif des Services Web Basé sur les MEFE", *7ème Conférence Internationale sur les NOuvelles TEchnologies de la REpartition (NOTERE)*, pub: Hermès. Marrakech, Maroc, 2007, pp. 329-340.

[5] A. Benharref, M. A. Serhani, R. Dssouli, and R. Glitho, "Les Agents Mobiles pour la Vérification de la QdS des Services Web", *6ème Conférence Internationale sur les NOuvelles TEchnologies de la REpartition (NOTERE)*, pub: Lavoisier. Toulouse, France, 2006, pp. 339-350.

[6] A. Benharref, R. Dssouli, R. Glitho, and M. A. Serhani, "Towards the testing of composed Web Services in 3rd generation networks", *IFIP International Conference on Testing of Communicating Systems (TestCom), Lecture Notes in Computer Science*, vol. 3964, pub: Springer Verlag. New York, NY, United States, 2006, pp. 118-133.

[7] A. Benharref, M. A. Serhani, R. Glitho, and R. Dssouli, "Une architecture Multi-Observateur pour l'Observation des Services Web Composés", *5ème Conférence Internationale sur les NOuvelles TEchnologies de la REpartition (NOTERE)*. Gatineau, Quebec, Canada, 2005, pp. 153-162.

[8] A. Benharref, R. Glitho, and R. Dssouli, "A Web Service Based-Architecture for Detecting Faults in Web Services", *Integrated Management*, pub: IEEE Press. Nice, France, 2005.

[9] A. Benharref, R. Glitho, and R. Dssouli, "Mobile agents for testing Web Services in Next Generation Networks", *MATA 2005, Lecture Notes in Computer Science*, vol. 3744, pub: Springer Verlag. Montreal, Canada, 2005, pp. 182-191.

[10] M. A. Serhani, R. Dssouli, H. Sahraoui, A. Benharref, and M. E. Badidi, "VAQoS: architecture for end-to-end QoS management of value added Web Services," *International Journal of Intelligent Information Technologies*, vol. 2, pp. 37-56, 2006.

[11] M.A. Serhani, R. Dssouli, A. Benharef, H. Sahraoui, E. Badidi "Toward integration of Quality of Web services (QoWS) Management operations in SOA: Case of basic and composite Web services," selected as a Book Chapter in Advanced Topics in Intelligent Information Technologies.

[12] M. A. Serhani, A. Benharref, R. Dssouli, and H. Sahraoui, "CompQoS: Towards an Architecture for QoS composition and monitoring (validation) of composite Web Services", *International Conference on Web Technologies, Application, and Services "WTAS"*. Calgary, Canada, 2006, pp. 78-83.

[13] M. A. Serhani, R. Dssouli, H. Sahraoui, A. Hafid, and A. Benharref, "Toward a new Web Services development life cycle", *International Multi-Conferences in Computer Science & Computer Engineering, International Symposium on Web Services and Applications*. Las Vegas, Nevada, USA, 2005, pp. 94-103.

[14] M. A. Serhani, R. Dssouli, H. Sahraoui, A. Benharref, and M. E. Badidi, "QoS Integration in Value Added Web Services", *2nd international conference on Innovations in Information Technology*. Dubai, U.A.E, 2005.

[15] M. A. Serhani, A. Hafid, S. Houari, and A. Benharref, "QoS Broker- Based Architecture for Web Services," *NOuvelles TEchnologies de la REpartition (NOTERE)"*, Saîdia, Morocco, 2004, pp. 68-81.

[16] M.A. Serhani, M. V. Salem, R. Dssouli, A. Benharref, E. Badidi , "A Multi-Broker based Architecture for QoS-aware Web services Selection and QoS Management," In preparation.

We live on an island surrounded by a sea of ignorance. As our island of knowledge grows, so does the shore of our ignorance.

John A. Wheeler